Celery Documentation

Release 2.4.7

Ask Solem
Contributors

February 04, 2014

Contents:

CHAPTER 1

Getting Started

Release 2.4

Date February 04, 2014

1.1 Introduction

Version 2.4.7

Web http://celeryproject.org/

Download http://pypi.python.org/pypi/celery/

Source http://github.com/ask/celery/

Keywords task queue, job queue, asynchronous, rabbitmq, amqp, redis, python, webhooks, queue, distributed

– Celery is an open source asynchronous task queue/job queue based on distributed message passing. It is focused on real-time operation, but supports scheduling as well.

The execution units, called tasks, are executed concurrently on one or more worker nodes using multiprocessing, Eventlet or gevent. Tasks can execute asynchronously (in the background) or synchronously (wait until ready).

Celery is used in production systems to process millions of tasks a day.

Celery is written in Python, but the protocol can be implemented in any language. It can also operate with other languages using webhooks.

The recommended message broker is RabbitMQ, but limited support for Redis, Beanstalk, MongoDB, CouchDB and databases (using SQLAlchemy or the Django ORM) is also available.

Celery is easy to integrate with Django, Pylons and Flask, using the django-celery, celery-pylons and Flask-Celery add-on packages.

- Overview
- Example
- Features
- Documentation
- Installation
  - Bundles
  - Downloading and installing from source
  - Using the development version

1.1.1 Overview

This is a high level overview of the architecture.

The broker delivers tasks to the worker nodes. A worker node is a networked machine running *celeryd*. This can be one or more machines depending on the workload.

The result of the task can be stored for later retrieval (called its "tombstone").

1.1.2 Example

You probably want to see some code by now, so here's an example task adding two numbers:

```
from celery.task import task

@task
def add(x, y):
    return x + y
```

You can execute the task in the background, or wait for it to finish:

```
>>> result = add.delay(4, 4)
>>> result.wait() # wait for and return the result
8
```

Simple!

1.1.3 Features

Messaging	Supported brokers include RabbitMQ, Redis, Beanstalk, MongoDB, CouchDB, and popular SQL databases.
Fault-tolerant	Excellent configurable error recovery when using *RabbitMQ*, ensures your tasks are never lost.
Distributed	Runs on one or more machines. Supports broker clustering and HA when used in combination with RabbitMQ. You can set up new workers without central configuration (e.g. use your grandma's laptop to help if the queue is temporarily congested).
Concurrency	Concurrency is achieved by using multiprocessing, Eventlet, *gevent* or a mix of these.
Scheduling	Supports recurring tasks like cron, or specifying an exact date or countdown for when after the task should be executed.
Latency	Low latency means you are able to execute tasks *while the user is waiting*.
Return Values	Task return values can be saved to the selected result store backend. You can wait for the result, retrieve it later, or ignore it.
Result Stores	Database, MongoDB, Redis, *Tokyo Tyrant*, *Cassandra*, or AMQP (message notification).
Webhooks	Your tasks can also be HTTP callbacks, enabling cross-language communication.
Rate limiting	Supports rate limiting by using the token bucket algorithm, which accounts for bursts of traffic. Rate limits can be set for each task type, or globally for all.
Routing	Using AMQP's flexible routing model you can route tasks to different workers, or select different message topologies, by configuration or even at runtime.
Remote-control	Worker nodes can be controlled from remote by using broadcast messaging. A range of built-in commands exist in addition to the ability to easily define your own. (AMQP/Redis only)
Monitoring	You can capture everything happening with the workers in real-time by subscribing to events. A real-time web monitor is in development.
Serialization	Supports Pickle, JSON, YAML, or easily defined custom schemes. One task invocation can have a different scheme than another.
Tracebacks	Errors and tracebacks are stored and can be investigated after the fact.
UUID	Every task has an UUID (Universally Unique Identifier), which is the task id used to query task status and return value.
Retries	Tasks can be retried if they fail, with configurable maximum number of retries, and delays between each retry.
Task Sets	A Task set is a task consisting of several sub-tasks. You can find out how many, or if all of the sub-tasks has been executed, and even retrieve the results in order. Progress bars, anyone?
Made for Web	You can query status and results via URLs, enabling the ability to poll task status using Ajax.
Error Emails	Can be configured to send emails to the administrators when tasks fails.

1.1.4 Documentation

The latest documentation with user guides, tutorials and API reference is hosted at Github.

1.1.5 Installation

You can install Celery either via the Python Package Index (PyPI) or from source.

To install using *pip*,:

```
$ pip install -U Celery
```

To install using *easy_install*,:

```
$ easy_install -U Celery
```

Bundles

Celery also defines a group of bundles that can be used to install Celery and the dependencies for a given feature.

The following bundles are available:

celery-with-redis for using Redis as a broker.

celery-with-mongodb for using MongoDB as a broker.

django-celery-with-redis for Django, and using Redis as a broker.

django-celery-with-mongodb for Django, and using MongoDB as a broker.

bundle-celery convenience bundle installing *Celery* and related packages.

Downloading and installing from source

Download the latest version of Celery from http://pypi.python.org/pypi/celery/

You can install it by doing the following,:

```
$ tar xvfz celery-0.0.0.tar.gz
$ cd celery-0.0.0
$ python setup.py build
# python setup.py install # as root
```

Using the development version

You can clone the repository by doing the following:

```
$ git clone git://github.com/ask/celery.git
```

1.2 Brokers

Release 2.4

Date February 04, 2014

Celery supports several message transport alternatives.

1.2.1 Using RabbitMQ

- Installation & Configuration
- Installing the RabbitMQ Server
  - Setting up RabbitMQ
  - Installing RabbitMQ on OS X
    - Configuring the system host name
    - Starting/Stopping the RabbitMQ server

Installation & Configuration

RabbitMQ is the default broker so it does not require any additional dependencies or initial configuration, other than the URL location of the broker instance you want to use:

```
>>> BROKER_URL = "amqp://guest:guest@localhost:5672//"
```

For a description of broker URLs and a full list of the various broker configuration options available to Celery, see *Broker Settings*.

Installing the RabbitMQ Server

See Installing RabbitMQ over at RabbitMQ's website. For Mac OS X see Installing RabbitMQ on OS X.

Note: If you're getting *nodedown* errors after installing and using **rabbitmqctl** then this blog post can help you identify the source of the problem:

http://somic.org/2009/02/19/on-rabbitmqctl-and-badrpcnodedown/

Setting up RabbitMQ

To use celery we need to create a RabbitMQ user, a virtual host and allow that user access to that virtual host:

```
$ rabbitmqctl add_user myuser mypassword

$ rabbitmqctl add_vhost myvhost

$ rabbitmqctl set_permissions -p myvhost myuser ".*" ".*" ".*"
```

See the RabbitMQ Admin Guide for more information about access control.

Installing RabbitMQ on OS X

The easiest way to install RabbitMQ on Snow Leopard is using Homebrew; the new and shiny package management system for OS X.

In this example we'll install Homebrew into `/lol`, but you can choose whichever destination, even in your home directory if you want, as one of the strengths of Homebrew is that it's relocatable.

Homebrew is actually a git repository, so to install Homebrew, you first need to install git. Download and install from the disk image at http://code.google.com/p/git-osx-installer/downloads/list?can=3

When git is installed you can finally clone the repository, storing it at the `/lol` location:

```
$ git clone git://github.com/mxcl/homebrew /lol
```

Brew comes with a simple utility called **brew**, used to install, remove and query packages. To use it you first have to add it to `PATH`, by adding the following line to the end of your `~/.profile`:

```
export PATH="/lol/bin:/lol/sbin:$PATH"
```

Save your profile and reload it:

```
$ source ~/.profile
```

Finally, we can install rabbitmq using **brew**:

```
$ brew install rabbitmq
```

Configuring the system host name If you're using a DHCP server that is giving you a random host name, you need to permanently configure the host name. This is because RabbitMQ uses the host name to communicate with nodes.

Use the **scutil** command to permanently set your host name:

```
sudo scutil --set HostName myhost.local
```

Then add that host name to `/etc/hosts` so it's possible to resolve it back into an IP address:

```
127.0.0.1       localhost myhost myhost.local
```

If you start the rabbitmq server, your rabbit node should now be *rabbit@myhost*, as verified by **rabbitmqctl**:

```
$ sudo rabbitmqctl status
Status of node rabbit@myhost ...
[{running_applications,[{rabbit,"RabbitMQ","1.7.1"},
                        {mnesia,"MNESIA  CXC 138 12","4.4.12"},
                        {os_mon,"CPO  CXC 138 46","2.2.4"},
                        {sasl,"SASL  CXC 138 11","2.1.8"},
                        {stdlib,"ERTS  CXC 138 10","1.16.4"},
                        {kernel,"ERTS  CXC 138 10","2.13.4"}]},
{nodes,[rabbit@myhost]},
{running_nodes,[rabbit@myhost]}]
...done.
```

This is especially important if your DHCP server gives you a host name starting with an IP address, (e.g. *23.10.112.31.comcast.net*), because then RabbitMQ will try to use *rabbit@23*, which is an illegal host name.

Starting/Stopping the RabbitMQ server To start the server:

```
$ sudo rabbitmq-server
```

you can also run it in the background by adding the `-detached` option (note: only one dash):

```
$ sudo rabbitmq-server -detached
```

Never use **kill** to stop the RabbitMQ server, but rather use the **rabbitmqctl** command:

```
$ sudo rabbitmqctl stop
```

When the server is running, you can continue reading Setting up RabbitMQ.

1.2.2 Using Redis

Installation

For the Redis support you have to install additional dependencies. You can install both Celery and these dependencies in one go using ehter the celery-with-redis, or the *django-celery-with-redis* bundles:

```
$ pip install -U celery-with-redis
```

Configuration

Configuration is easy, set the transport, and configure the location of your Redis database:

```
BROKER_URL = "redis://localhost:6379/0"
```

Where the URL is in the format of:

```
redis://userid:password@hostname:port/db_number
```

Results

If you also want to store the state and return values of tasks in Redis, you should configure these settings:

```
CELERY_RESULT_BACKEND = "redis"
CELERY_REDIS_HOST = "localhost"
CELERY_REDIS_PORT = 6379
CELERY_REDIS_DB = 0
```

For a complete list of options supported by the Redis result backend see *Redis backend settings*

1.2.3 Using SQLAlchemy

Installation

For the SQLAlchemy transport you have to install the *kombu-sqlalchemy* library:

```
$ pip install -U kombu-sqlalchemy
```

Configuration

This transport uses only the `BROKER_HOST` setting, which have to be an SQLAlchemy database URI.

1. Set your broker transport:

   ```
   BROKER_TRANSPORT = "sqlalchemy"
   ```

2. Configure the database URI:

   ```
   BROKER_HOST = "sqlite:///celerydb.sqlite"
   ```

Please see SQLAlchemy: Supported Databases for a table of supported databases. Some other SQLAlchemy Connection String, examples:

```
# sqlite (filename)
BROKER_HOST = "sqlite:///celerydb.sqlite"

# mysql
BROKER_HOST = "mysql://scott:tiger@localhost/foo"

# postgresql
BROKER_HOST = "postgresql://scott:tiger@localhost/mydatabase"

# oracle
BROKER_HOST = "oracle://scott:tiger@127.0.0.1:1521/sidname"
```

Results

To store results in the database as well, you should configure the result backend. See *Database backend settings*.

Limitations

The SQLAlchemy database transport does not currently support:

- Remote control commands (celeryev, broadcast)
- Events, including the Django Admin monitor.
- Using more than a few workers (can lead to messages being executed multiple times).

1.2.4 Using the Django Database

Installation

For the Django database transport support you have to install the *django-kombu* library:

```
$ pip install -U django-kombu
```

Configuration

The database transport uses the Django *DATABASE_** settings for database configuration values.

1. Set your broker transport:

   ```
   BROKER_URL = "django://"
   ```

2. Add `djkombu` to *INSTALLED_APPS*:

   ```
   INSTALLED_APPS = ("djkombu", )
   ```

3. Verify your database settings:

   ```
   DATABASE_ENGINE = "mysql"
   DATABASE_NAME = "mydb"
   DATABASE_USER = "myuser"
   DATABASE_PASSWORD = "secret"
   ```

   The above is just an example, if you haven't configured your database before you should read the Django database settings reference: http://docs.djangoproject.com/en/1.1/ref/settings/#database-engine

1. Sync your database schema:

```
$ python manage.py syncdb
```

Limitations

The Django database transport does not currently support:

- Remote control commands (celeryev, broadcast)
- Events, including the Django Admin monitor.
- Using more than a few workers (can lead to messages being executed multiple times).

1.2.5 Using MongoDB

Installation

For the MongoDB support you have to install additional dependencies. You can install both Celery and these dependencies in one go using either the celery-with-mongodb, or the *django-celery-with-mongodb* bundles:

```
$ pip install -U celery-with-mongodb
```

Configuration

Configuration is easy, set the transport, and configure the location of your MongoDB database:

```
BROKER_URL = "mongodb://localhost:27017/database_name"
```

Where the URL is in the format of:

```
mongodb://userid:password@hostname:port/database_name
```

The host name will default to `localhost` and the port to 27017, and so they are optional. userid and password are also optional, but needed if your MongoDB server requires authentication.

Results

If you also want to store the state and return values of tasks in MongoDB, you should see *MongoDB backend settings*.

Limitations

The mongodb message transport currently does not support:

- Remote control commands (celeryctl, broadcast)

1.2.6 Using CouchDB

Installation

For the CouchDB support you have to install additional dependencies. You can install both Celery and these dependencies in one go using either the celery-with-couchdb, or the *django-celery-with-couchdb* bundles:

```
$ pip install -U celery-with-couchdb
```

Configuration

Configuration is easy, set the transport, and configure the location of your CouchDB database:

```
BROKER_URL = "couchdb://localhost:5984/database_name"
```

Where the URL is in the format of:

```
couchdb://userid:password@hostname:port/database_name
```

The host name will default to `localhost` and the port to 5984, and so they are optional. userid and password are also optional, but needed if your CouchDB server requires authentication.

Results

Storing task state and results in CouchDB is currently **not supported**.

Limitations

The Beanstalk message transport does not currently support:

- Remote control commands (celeryctl, broadcast)

1.2.7 Using Beanstalk

Installation

For the Beanstalk support you have to install additional dependencies. You can install both Celery and these dependencies in one go using either the celery-with-beanstalk, or the *django-celery-with-beanstalk* bundles:

```
$ pip install -U celery-with-beanstalk
```

Configuration

Configuration is easy, set the transport, and configure the location of your CouchDB database:

```
BROKER_URL = "beanstalk://localhost:11300"
```

Where the URL is in the format of:

```
beanstalk://hostname:port
```

The host name will default to `localhost` and the port to 11300, and so they are optional.

Results

Using Beanstalk to store task state and results is currently **not supported**.

Limitations

The Beanstalk message transport does not currently support:

- Remote control commands (celeryctl, broadcast)
- Authentication

1.3 First steps with Celery

- Choosing your Broker
- Creating a simple task
- Configuration
- Running the celery worker server
- Executing the task
  - Keeping Results
- Where to go from here

1.3.1 Choosing your Broker

Before you can use Celery you need to choose, install and run a broker. The broker is the service responsible for receiving and delivering task messages.

There are several choices available, including:

- *Using RabbitMQ*

RabbitMQ is feature-complete, safe and durable. If not losing tasks is important to you, then this is your best option.

- *Using Redis*

Redis is also feature-complete, but power failures or abrupt termination may result in data loss.

- *Using SQLAlchemy*
- *Using the Django Database*

Using a database as a message queue is not recommended, but can be sufficient for very small installations. Celery can use the SQLAlchemy and Django ORM.

- and more.

In addition to the above, there are several other transport implementations to choose from, including *Using CouchDB*, *Using Beanstalk*, *Using MongoDB*, and SQS. There is a Transport Comparison in the Kombu documentation.

1.3.2 Creating a simple task

In this tutorial we are creating a simple task that adds two numbers. Tasks are defined in normal Python modules.

By convention we will call our module `tasks.py`, and it looks like this:

file *tasks.py*

```
from celery.task import task

@task
def add(x, y):
    return x + y
```

Behind the scenes the *@task* decorator actually creates a class that inherits from `Task`. The best practice is to only create custom task classes when you want to change generic behavior, and use the decorator to define tasks.

See also:

The full documentation on how to create tasks and task classes is in the *Tasks* part of the user guide.

1.3.3 Configuration

Celery is configured by using a configuration module. By default this module is called `celeryconfig.py`.

The configuration module must either be in the current directory or on the Python path, so that it can be imported.

You can also set a custom name for the configuration module by using the `CELERY_CONFIG_MODULE` environment variable.

Let's create our `celeryconfig.py`.

1. Configure how we communicate with the broker (RabbitMQ in this example):

   ```
   BROKER_URL = "amqp://guest:guest@localhost:5672//"
   ```

2. Define the backend used to store task metadata and return values:

   ```
   CELERY_RESULT_BACKEND = "amqp"
   ```

   The AMQP backend is non-persistent by default, and you can only fetch the result of a task once (as it's sent as a message).

   For list of backends available and related options see *Task result backend settings*.

3. Finally we list the modules the worker should import. This includes the modules containing your tasks.

   We only have a single task module, `tasks.py`, which we added earlier:

   ```
   CELERY_IMPORTS = ("tasks", )
   ```

That's it.

There are more options available, like how many processes you want to use to process work in parallel (the `CELERY_CONCURRENCY` setting), and we could use a persistent result store backend, but for now, this should do. For all of the options available, see *Configuration and defaults*.

Note: You can also specify modules to import using the `-I` option to `celeryd`:

```
$ celeryd -l info -I tasks,handlers
```

This can be a single, or a comma separated list of task modules to import when **celeryd** starts.

1.3.4 Running the celery worker server

To test we will run the worker server in the foreground, so we can see what's going on in the terminal:

```
$ celeryd --loglevel=INFO
```

In production you will probably want to run the worker in the background as a daemon. To do this you need to use the tools provided by your platform, or something like supervisord (see *Running celeryd as a daemon* for more information).

For a complete listing of the command line options available, do:

```
$  celeryd --help
```

1.3.5 Executing the task

Whenever we want to execute our task, we use the `delay()` method of the task class.

This is a handy shortcut to the `apply_async()` method which gives greater control of the task execution (see *Executing Tasks*).

```
>>> from tasks import add
>>> add.delay(4, 4)
<AsyncResult: 889143a6-39a2-4e52-837b-d80d33efb22d>
```

At this point, the task has been sent to the message broker. The message broker will hold on to the task until a worker server has consumed and executed it.

Right now we have to check the worker log files to know what happened with the task. Applying a task returns an `AsyncResult`, if you have configured a result store the `AsyncResult` enables you to check the state of the task, wait for the task to finish, get its return value or exception/traceback if the task failed, and more.

Keeping Results

If you want to keep track of the tasks state, Celery needs to store or send the states somewhere. There are several built-in backends to choose from: SQLAlchemy/Django ORM, Memcached, Redis, AMQP, MongoDB, Tokyo Tyrant and Redis – or you can define your own.

For this example we will use the *amqp* result backend, which sends states as messages. The backend is configured via the `CELERY_RESULT_BACKEND` option, in addition individual result backends may have additional settings you can configure:

```
CELERY_RESULT_BACKEND = "amqp"

#: We want the results to expire in 5 minutes, note that this requires
#: RabbitMQ version 2.1.1 or higher, so please comment out if you have
#: an earlier version.
CELERY_TASK_RESULT_EXPIRES = 300
```

To read more about result backends please see *Result Backends*.

Now with the result backend configured, let's execute the task again. This time we'll hold on to the `AsyncResult`:

```
>>> result = add.delay(4, 4)
```

Here's some examples of what you can do when you have results:

```
>>> result.ready() # returns True if the task has finished processing.
False

>>> result.result # task is not ready, so no return value yet.
None
```

```
>>> result.get()    # Waits until the task is done and returns the retval.
8

>>> result.result # direct access to result, doesn't re-raise errors.
8

>>> result.successful() # returns True if the task didn't end in failure.
True
```

If the task raises an exception, the return value of *result.successful()* will be `False`, and *result.result* will contain the exception instance raised by the task.

1.3.6 Where to go from here

After this you should read the *User Guide*. Specifically *Tasks* and *Executing Tasks*.

1.4 Resources

- Getting Help
  - Mailing list
  - IRC
- Bug tracker
- Wiki
- Contributing
- License

1.4.1 Getting Help

Mailing list

For discussions about the usage, development, and future of celery, please join the celery-users mailing list.

IRC

Come chat with us on IRC. The #celery channel is located at the Freenode network.

1.4.2 Bug tracker

If you have any suggestions, bug reports or annoyances please report them to our issue tracker at http://github.com/ask/celery/issues/

1.4.3 Wiki

http://wiki.github.com/ask/celery/

1.4.4 Contributing

Development of *celery* happens at Github: http://github.com/ask/celery

You are highly encouraged to participate in the development of *celery*. If you don't like Github (for some reason) you're welcome to send regular patches.

Be sure to also read the Contributing to Celery section in the documentation.

1.4.5 License

This software is licensed under the *New BSD License*. See the `LICENSE` file in the top distribution directory for the full license text.

CHAPTER 2

User Guide

Release 2.4

Date February 04, 2014

2.1 Overview

- Tasks
- Workers
- Monitoring
- Routing
- Ecosystem
  - Kombu
  - cyme
  - celerymon

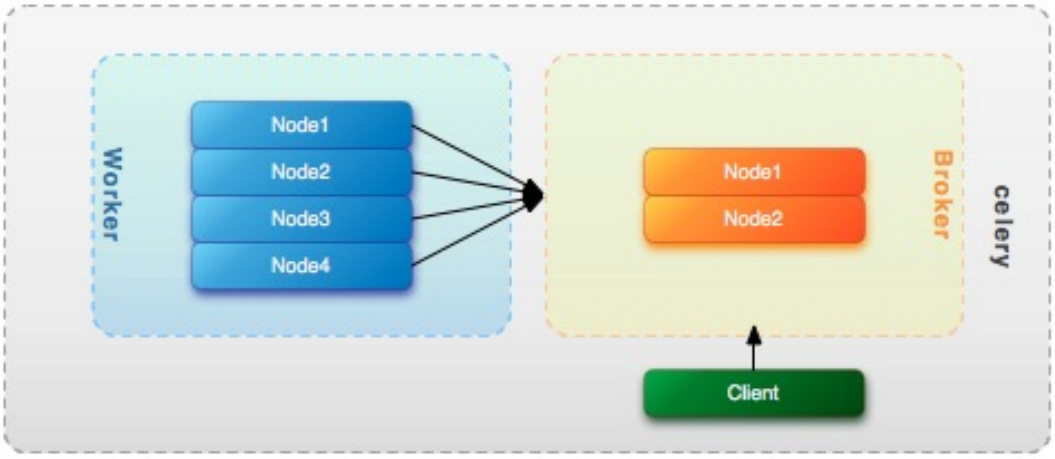


Figure 2.1: *Figure 1:* Worker and broker nodes.

To use Celery you need at least two main components; a message broker and a worker.

The message broker enables clients and workers to communicate through messaging. There are several broker implementations available, the most popular being RabbitMQ.

The worker processes messages, and consists of one or more physical (or virtual) nodes.

2.1.1 Tasks

The action to take whenever a message of a certain type is received is called a "task".

- Go to *Tasks*.
- Go to *Executing Tasks*.
- Go to *Sets of tasks, Subtasks and Callbacks*
- Go to *Periodic Tasks*.
- Go to *HTTP Callback Tasks (Webhooks)*.

2.1.2 Workers

Go to *Workers Guide*.

2.1.3 Monitoring

Go to *Monitoring and Management Guide*.

2.1.4 Routing

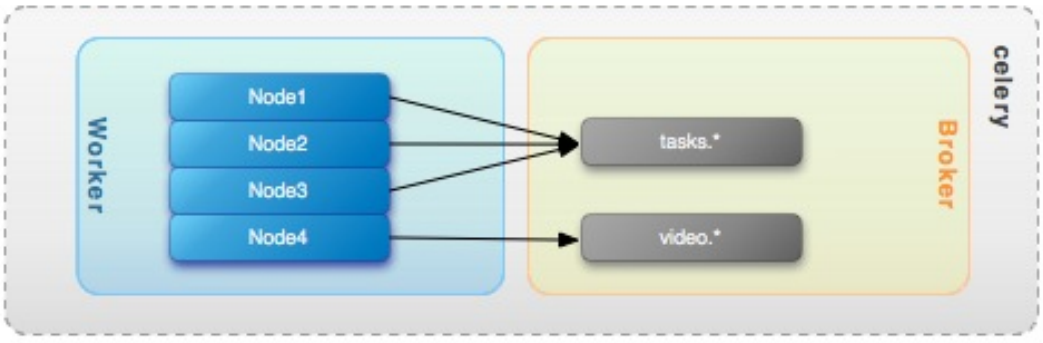


Figure 2.2: *Figure 2:* Worker bindings.

Go to *Routing Tasks*.

Celery takes advantage of AMQPs flexible routing model. Tasks can be routed to specific servers, or a cluster of servers by binding workers to different queues. A single worker node can be bound to one or more queues. Multiple messaging scenarios are supported: round robin, point-to-point, broadcast (one-to-many), and more.

Celery aims to hide the complexity of AMQP through features like *Automatic routing*, while still preserving the ability to go low level if that should be necessary.

2.1.5 Ecosystem

Kombu

cyme

celerymon

2.2 Tasks

- Basics
- Context
  - Example Usage
- Logging
- Retrying a task if something fails
  - Using a custom retry delay
- Task options
  - General
  - Message and routing options
- Task names
  - Automatic naming and relative imports
- Decorating tasks
- Task States
  - Result Backends
    - AMQP Result Backend
    - Database Result Backend
  - Built-in States
    - PENDING
    - STARTED
    - SUCCESS
    - FAILURE
    - RETRY
    - REVOKED
  - Custom states
  - Creating pickleable exceptions
- Creating custom task classes
  - Instantiation
  - Abstract classes
  - Handlers
    - on_retry
- How it works
- Tips and Best Practices
  - Ignore results you don't want
  - Disable rate limits if they're not used
  - Avoid launching synchronous subtasks
- Performance and Strategies
  - Granularity
  - Data locality
  - State
  - Database transactions
- Example
  - blog/models.py
  - blog/views.py
  - blog/tasks.py

This guide gives an overview of how tasks are defined. For a complete listing of task attributes and methods, please see the `API reference`.

2.2.1 Basics

A task is a class that encapsulates a function and its execution options. Given a function create_user‘, that takes two arguments: *username* and *password*, you can create a task like this:

```
from django.contrib.auth import User

from celery.task import task

@task
def create_user(username, password):
    User.objects.create(username=username, password=password)
```

Task options are added as arguments to *task*:

```
@task(serializer="json")
def create_user(username, password):
    User.objects.create(username=username, password=password)
```

2.2.2 Context

task.request contains information and state related the currently executing task, and must always contain the following attributes:

id The unique id of the executing task.

taskset The unique id of the taskset this task is a member of (if any).

args Positional arguments.

kwargs Keyword arguments.

retries How many times the current task has been retried. An integer starting at *0*.

is_eager Set to `True` if the task is executed locally in the client, and not by a worker.

logfile The file the worker logs to. See Logging.

loglevel The current log level used.

hostname Hostname of the worker instance executing the task.

delivery_info

Additional message delivery information. This is a mapping containing the exchange and routing key used to deliver this task. Used by e.g. `retry()` to resend the task to the same destination queue.

NOTE As some messaging backends don't have advanced routing capabilities, you can't trust the availability of keys in this mapping.

Example Usage

```
from celery.task import task

@task
def add(x, y):
    print("Executing task id %r, args: %r kwargs: %r" % (
        add.request.id, add.request.args, add.request.kwargs))
```

2.2.3 Logging

You can use the workers logger to add diagnostic output to the worker log:

```
@task
def add(x, y):
    logger = add.get_logger()
    logger.info("Adding %s + %s" % (x, y))
    return x + y
```

There are several logging levels available, and the workers *loglevel* setting decides whether or not they will be written to the log file.

Of course, you can also simply use *print* as anything written to standard out/-err will be written to the log file as well.

2.2.4 Retrying a task if something fails

Simply use `retry()` to re-send the task. It will do the right thing, and respect the `max_retries` attribute:

```
@task
def send_twitter_status(oauth, tweet):
    try:
        twitter = Twitter(oauth)
        twitter.update_status(tweet)
    except (Twitter.FailWhaleError, Twitter.LoginError), exc:
        send_twitter_status.retry(exc=exc)
```

Here we used the *exc* argument to pass the current exception to `retry()`. At each step of the retry this exception is available as the tombstone (result) of the task. When `max_retries` has been exceeded this is the exception raised. However, if an *exc* argument is not provided the `RetryTaskError` exception is raised instead.

Note: The `retry()` call will raise an exception so any code after the retry will not be reached. This is the `celery.exceptions.RetryTaskError` exception, it is not handled as an error but rather as a semi-predicate to signify to the worker that the task is to be retried.

This is normal operation and always happens unless the `throw` argument to retry is set to `False`.

Using a custom retry delay

When a task is to be retried, it will wait for a given amount of time before doing so. The default delay is in the `default_retry_delay` attribute on the task. By default this is set to 3 minutes. Note that the unit for setting the delay is in seconds (int or float).

You can also provide the *countdown* argument to `retry()` to override this default.

```
@task(default_retry_delay=30 * 60)  # retry in 30 minutes.
def add(x, y):
    try:
        ...
    except Exception, exc:
        add.retry(exc=exc, countdown=60)  # override the default and
                                          # retry in 1 minute
```

2.2.5 Task options

General

`Task`.**`name`**
The name the task is registered as.

You can set this name manually, or just use the default which is automatically generated using the module and class name. See *Task names*.

`Task`.**`abstract`**
Abstract classes are not registered, but are used as the base class for new task types.

`Task`.**`max_retries`**
The maximum number of attempted retries before giving up. If this exceeds the `MaxRetriesExceeded` an exception will be raised. *NOTE:* You have to `retry()` manually, it's not something that happens automatically.

`Task`.**`default_retry_delay`**
Default time in seconds before a retry of the task should be executed. Can be either `int` or `float`. Default is a 3 minute delay.

`Task`.**`rate_limit`**
Set the rate limit for this task type, i.e. how many times in a given period of time is the task allowed to run.

If this is `None` no rate limit is in effect. If it is an integer, it is interpreted as "tasks per second".

The rate limits can be specified in seconds, minutes or hours by appending *"/s"*, *"/m"* or *"/h"* to the value. Example: *"100/m"* (hundred tasks a minute). Default is the `CELERY_DEFAULT_RATE_LIMIT` setting, which if not specified means rate limiting for tasks is disabled by default.

`Task`.**`time_limit`**
The hard time limit for this task. If not set then the workers default will be used.

`Task`.**`soft_time_limit`**
The soft time limit for this task. If not set then the workers default will be used.

`Task`.**`ignore_result`**
Don't store task state. Note that this means you can't use `AsyncResult` to check if the task is ready, or get its return value.

`Task`.**`store_errors_even_if_ignored`**
If `True`, errors will be stored even if the task is configured to ignore results.

`Task`.**`send_error_emails`**
Send an email whenever a task of this type fails. Defaults to the `CELERY_SEND_TASK_ERROR_EMAILS` setting. See *Error E-Mails* for more information.

`Task`.**`error_whitelist`**
If the sending of error emails is enabled for this task, then this is a white list of exceptions to actually send emails about.

`Task`.**`serializer`**
A string identifying the default serialization method to use. Defaults to the `CELERY_TASK_SERIALIZER` setting. Can be *pickle* *json*, *yaml*, or any custom serialization methods that have been registered with `kombu.serialization.registry`.

Please see *Serializers* for more information.

`Task`.**`backend`**
The result store backend to use for this task. Defaults to the `CELERY_RESULT_BACKEND` setting.

`Task.`**`acks_late`**
If set to `True` messages for this task will be acknowledged **after** the task has been executed, not *just before*, which is the default behavior.

Note that this means the task may be executed twice if the worker crashes in the middle of execution, which may be acceptable for some applications.

The global default can be overridden by the `CELERY_ACKS_LATE` setting.

`Task.`**`track_started`**
If `True` the task will report its status as "started" when the task is executed by a worker. The default value is `False` as the normal behaviour is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried. Having a "started" status can be useful for when there are long running tasks and there is a need to report which task is currently running.

The host name and process id of the worker executing the task will be available in the state metadata (e.g. *result.info["pid"]*)

The global default can be overridden by the `CELERY_TRACK_STARTED` setting.

See also:

The API reference for `BaseTask`.

Message and routing options

`Task.`**`queue`**
Use the routing settings from a queue defined in `CELERY_QUEUES`. If defined the `exchange` and `routing_key` options will be ignored.

`Task.`**`exchange`**
Override the global default *exchange* for this task.

`Task.`**`routing_key`**
Override the global default *routing_key* for this task.

`Task.`**`mandatory`**
If set, the task message has mandatory routing. By default the task is silently dropped by the broker if it can't be routed to a queue. However – If the task is mandatory, an exception will be raised instead.

Not supported by amqplib.

`Task.`**`immediate`**
Request immediate delivery. If the task cannot be routed to a task worker immediately, an exception will be raised. This is instead of the default behavior, where the broker will accept and queue the task, but with no guarantee that the task will ever be executed.

Not supported by amqplib.

`Task.`**`priority`**
The message priority. A number from 0 to 9, where 0 is the highest priority.

Not supported by RabbitMQ.

See also:

Routing options for more information about message options, and *Routing Tasks*.

2.2.6 Task names

The task type is identified by the *task name*.

If not provided a name will be automatically generated using the module and class name.

For example:

```
>>> @task(name="sum-of-two-numbers")
>>> def add(x, y):
...     return x + y

>>> add.name
'sum-of-two-numbers'
```

The best practice is to use the module name as a prefix to classify the tasks using namespaces. This way the name won't collide with the name from another module:

```
>>> @task(name="tasks.add")
>>> def add(x, y):
...     return x + y

>>> add.name
'tasks.add'
```

Which is exactly the name that is automatically generated for this task if the module name is "tasks.py":

```
>>> @task()
>>> def add(x, y):
...     return x + y

>>> add.name
'tasks.add'
```

Automatic naming and relative imports

Relative imports and automatic name generation does not go well together, so if you're using relative imports you should set the name explicitly.

For example if the client imports the module "myapp.tasks" as ".tasks", and the worker imports the module as "myapp.tasks", the generated names won't match and an `NotRegistered` error will be raised by the worker.

This is also the case if using Django and using *project.myapp*:

```
INSTALLED_APPS = ("project.myapp", )
```

The worker will have the tasks registered as "project.myapp.tasks.*", while this is what happens in the client if the module is imported as "myapp.tasks":

```
>>> from myapp.tasks import add
>>> add.name
'myapp.tasks.add'
```

For this reason you should never use "project.app", but rather add the project directory to the Python path:

```
import os
import sys
sys.path.append(os.getcwd())

INSTALLED_APPS = ("myapp", )
```

This makes more sense from the reusable app perspective anyway.

2.2.7 Decorating tasks

When using other decorators you must make sure that the *task* decorator is applied last:

```
@task
@decorator2
@decorator1
def add(x, y):
    return x + y
```

Which means the *@task* decorator must be the top statement.

2.2.8 Task States

Celery can keep track of the tasks current state. The state also contains the result of a successful task, or the exception and traceback information of a failed task.

There are several *result backends* to choose from, and they all have different strengths and weaknesses (see *Result Backends*).

During its lifetime a task will transition through several possible states, and each state may have arbitrary metadata attached to it. When a task moves into a new state the previous state is forgotten about, but some transitions can be deducted, (e.g. a task now in the `FAILED` state, is implied to have been in the `STARTED` state at some point).

There are also sets of states, like the set of `failure states`, and the set of `ready states`.

The client uses the membership of these sets to decide whether the exception should be re-raised (`PROPAGATE_STATES`), or whether the result can be cached (it can if the task is ready).

You can also define *Custom states*.

Result Backends

Celery needs to store or send the states somewhere. There are several built-in backends to choose from: SQLAlchemy/Django ORM, Memcached, Redis, AMQP, MongoDB, Tokyo Tyrant and Redis – or you can define your own.

No backend works well for every use case. You should read about the strengths and weaknesses of each backend, and choose the most appropriate for your needs.

See also:

Task result backend settings

AMQP Result Backend

The AMQP result backend is special as it does not actually *store* the states, but rather sends them as messages. This is an important difference as it means that a result *can only be retrieved once*; If you have two processes waiting for the same result, one of the processes will never receive the result!

Even with that limitation, it is an excellent choice if you need to receive state changes in real-time. Using messaging means the client does not have to poll for new states.

There are several other pitfalls you should be aware of when using the AMQP backend:

- Every new task creates a new queue on the server, with thousands of tasks the broker may be overloaded with queues and this will affect performance in negative ways. If you're using RabbitMQ then each queue will be a separate Erlang process, so if you're planning to keep many results simultaneously you may have to increase the Erlang process limit, and the maximum number of file descriptors your OS allows.
- Old results will not be cleaned automatically, so you must make sure to consume the results or else the number of queues will eventually go out of control. If you're running RabbitMQ 2.1.1 or higher you can take advantage of the `x-expires` argument to queues, which will expire queues after a certain time limit after they are unused. The queue expiry can be set (in seconds) by the `CELERY_TASK_RESULT_EXPIRES` setting (not enabled by default).

For a list of options supported by the AMQP result backend, please see *AMQP backend settings*.

Database Result Backend

Keeping state in the database can be convenient for many, especially for web applications with a database already in place, but it also comes with limitations.

- Polling the database for new states is expensive, and so you should increase the polling intervals of operations such as *result.wait()*, and *tasksetresult.join()*
- Some databases use a default transaction isolation level that is not suitable for polling tables for changes.

  In MySQL the default transaction isolation level is *REPEATABLE-READ*, which means the transaction will not see changes by other transactions until the transaction is committed. It is recommended that you change to the *READ-COMMITTED* isolation level.

Built-in States

PENDING

Task is waiting for execution or unknown. Any task id that is not known is implied to be in the pending state.

STARTED

Task has been started. Not reported by default, to enable please see :attr‘Task.track_started‘.

metadata *pid* and *hostname* of the worker process executing the task.

SUCCESS

Task has been successfully executed.

metadata *result* contains the return value of the task.

propagates Yes

ready Yes

FAILURE

Task execution resulted in failure.

metadata *result* contains the exception occurred, and *traceback* contains the backtrace of the stack at the point when the exception was raised.

propagates Yes

RETRY

Task is being retried.

metadata *result* contains the exception that caused the retry, and *traceback* contains the backtrace of the stack at the point when the exceptions was raised.

propagates No

REVOKED

Task has been revoked.

propagates Yes

Custom states

You can easily define your own states, all you need is a unique name. The name of the state is usually an uppercase string. As an example you could have a look at `abortable tasks` which defines its own custom `ABORTED` state.

Use `Task.update_state` to update a task's state:

```
@task
def upload_files(filenames):
    for i, file in enumerate(filenames):
        upload_files.update_state(state="PROGRESS",
            meta={"current": i, "total": len(filenames)})
```

Here we created the state *"PROGRESS"*, which tells any application aware of this state that the task is currently in progress, and also where it is in the process by having *current* and *total* counts as part of the state metadata. This can then be used to create e.g. progress bars.

Creating pickleable exceptions

A little known Python fact is that exceptions must behave a certain way to support being pickled.

Tasks that raise exceptions that are not pickleable will not work properly when Pickle is used as the serializer.

To make sure that your exceptions are pickleable the exception *MUST* provide the original arguments it was instantiated with in its `.args` attribute. The simplest way to ensure this is to have the exception call `Exception.__init__`.

Let's look at some examples that work, and one that doesn't:

```
# OK:
class HttpError(Exception):
    pass

# BAD:
class HttpError(Exception):
```

```
    def __init__(self, status_code):
        self.status_code = status_code

# OK:
class HttpError(Exception):

    def __init__(self, status_code):
        self.status_code = status_code
        Exception.__init__(self, status_code)  # <-- REQUIRED
```

So the rule is: For any exception that supports custom arguments `*args`, `Exception.__init__(self, *args)` must be used.

There is no special support for *keyword arguments*, so if you want to preserve keyword arguments when the exception is unpickled you have to pass them as regular args:

```
class HttpError(Exception):

    def __init__(self, status_code, headers=None, body=None):
        self.status_code = status_code
        self.headers = headers
        self.body = body

        super(Exception, self).__init__(status_code, headers, body)
```

2.2.9 Creating custom task classes

All tasks inherit from the `celery.task.Task` class. The task's body is its `run()` method.

The following code,

```
@task
def add(x, y):
    return x + y
```

will do roughly this behind the scenes:

```
@task
def AddTask(Task):

    def run(self, x, y):
        return x + y
add = registry.tasks[AddTask.name]
```

Instantiation

A task is **not** instantiated for every request, but is registered in the task registry as a global instance.

This means that the `__init__` constructor will only be called once per process, and that the task class is semantically closer to an Actor.

If you have a task,

```
class NaiveAuthenticateServer(Task):

    def __init__(self):
        self.users = {"george": "password"}
```

```
    def run(self, username, password):
        try:
            return self.users[username] == password
        except KeyError:
            return False
```

And you route every request to the same process, then it will keep state between requests.

This can also be useful to keep cached resources:

```
class DatabaseTask(Task):
    _db = None

    @property
    def db(self):
        if self._db = None:
            self._db = Database.connect()
        return self._db
```

Abstract classes

Abstract classes are not registered, but are used as the base class for new task types.

```
class DebugTask(Task):
    abstract = True

    def after_return(self, \*args, \*\*kwargs):
        print("Task returned: %r" % (self.request, ))


@task(base=DebugTask)
def add(x, y):
    return x + y
```

Handlers

execute(self, request, pool, loglevel, logfile, **kw):

Parameters

- **request** – A `TaskRequest`.
- **pool** – The task pool.
- **loglevel** – Current loglevel.
- **logfile** – Name of the currently used logfile.
- **consumer** – The `Consumer`.

after_return (*self*, *status*, *retval*, *task_id*, *args*, *kwargs*, *einfo*)

Handler called after the task returns.

Parameters

- **status** – Current task state.
- **retval** – Task return value/exception.
- **task_id** – Unique id of the task.

- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – `ExceptionInfo` instance, containing the traceback (if any).

The return value of this handler is ignored.

on_failure(*self*, *exc*, *task_id*, *args*, *kwargs*, *einfo*)
This is run by the worker when the task fails.

Parameters

- **exc** – The exception raised by the task.
- **task_id** – Unique id of the failed task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – `ExceptionInfo` instance, containing the traceback.

The return value of this handler is ignored.

on_retry(*self*, *exc*, *task_id*, *args*, *kwargs*, *einfo*)
This is run by the worker when the task is to be retried.

Parameters

- **exc** – The exception sent to `retry()`.
- **task_id** – Unique id of the retried task.
- **args** – Original arguments for the retried task.
- **kwargs** – Original keyword arguments for the retried task.
- **einfo** – `ExceptionInfo` instance, containing the traceback.

The return value of this handler is ignored.

on_success(*self*, *retval*, *task_id*, *args*, *kwargs*)
Run by the worker if the task executes successfully.

Parameters

- **retval** – The return value of the task.
- **task_id** – Unique id of the executed task.
- **args** – Original arguments for the executed task.
- **kwargs** – Original keyword arguments for the executed task.

The return value of this handler is ignored.

on_retry

2.2.10 How it works

Here comes the technical details, this part isn't something you need to know, but you may be interested.

All defined tasks are listed in a registry. The registry contains a list of task names and their task classes. You can investigate this registry yourself:

```
>>> from celery import registry
>>> from celery import task
>>> registry.tasks
{'celery.delete_expired_task_meta':
    <PeriodicTask: celery.delete_expired_task_meta (periodic)>,
 'celery.task.http.HttpDispatchTask':
    <Task: celery.task.http.HttpDispatchTask (regular)>,
 'celery.execute_remote':
    <Task: celery.execute_remote (regular)>,
 'celery.map_async':
    <Task: celery.map_async (regular)>,
 'celery.ping':
    <Task: celery.ping (regular)>}
```

This is the list of tasks built-in to celery. Note that we had to import *celery.task* first for these to show up. This is because the tasks will only be registered when the module they are defined in is imported.

The default loader imports any modules listed in the `CELERY_IMPORTS` setting.

The entity responsible for registering your task in the registry is a meta class, `TaskType`. This is the default meta class for `BaseTask`.

If you want to register your task manually you can mark the task as `abstract`:

```
class MyTask(Task):
    abstract = True
```

This way the task won't be registered, but any task inheriting from it will be.

When tasks are sent, we don't send any actual function code, just the name of the task to execute. When the worker then receives the message it can look up the name in its task registry to find the execution code.

This means that your workers should always be updated with the same software as the client. This is a drawback, but the alternative is a technical challenge that has yet to be solved.

2.2.11 Tips and Best Practices

Ignore results you don't want

If you don't care about the results of a task, be sure to set the `ignore_result` option, as storing results wastes time and resources.

```
@task(ignore_result=True)
def mytask(...)
    something()
```

Results can even be disabled globally using the `CELERY_IGNORE_RESULT` setting.

Disable rate limits if they're not used

Disabling rate limits altogether is recommended if you don't have any tasks using them. This is because the rate limit subsystem introduces quite a lot of complexity.

Set the `CELERY_DISABLE_RATE_LIMITS` setting to globally disable rate limits:

```
CELERY_DISABLE_RATE_LIMITS = True
```

Avoid launching synchronous subtasks

Having a task wait for the result of another task is really inefficient, and may even cause a deadlock if the worker pool is exhausted.

Make your design asynchronous instead, for example by using *callbacks*.

Bad:

```
@task
def update_page_info(url):
    page = fetch_page.delay(url).get()
    info = parse_page.delay(url, page).get()
    store_page_info.delay(url, info)

@task
def fetch_page(url):
    return myhttplib.get(url)

@task
def parse_page(url, page):
    return myparser.parse_document(page)

@task
def store_page_info(url, info):
    return PageInfo.objects.create(url, info)
```

Good:

```
@task(ignore_result=True)
def update_page_info(url):
    # fetch_page -> parse_page -> store_page
    fetch_page.delay(url, callback=subtask(parse_page,
                                           callback=subtask(store_page_info)))

@task(ignore_result=True)
def fetch_page(url, callback=None):
    page = myhttplib.get(url)
    if callback:
        # The callback may have been serialized with JSON,
        # so best practice is to convert the subtask dict back
        # into a subtask object.
        subtask(callback).delay(url, page)

@task(ignore_result=True)
def parse_page(url, page, callback=None):
    info = myparser.parse_document(page)
    if callback:
        subtask(callback).delay(url, info)

@task(ignore_result=True)
def store_page_info(url, info):
    PageInfo.objects.create(url, info)
```

We use `subtask` here to safely pass around the callback task. `subtask` is a subclass of dict used to wrap the arguments and execution options for a single task invocation.

See also:

Subtasks for more information about subtasks.

2.2.12 Performance and Strategies

Granularity

The task granularity is the amount of computation needed by each subtask. In general it is better to split the problem up into many small tasks, than have a few long running tasks.

With smaller tasks you can process more tasks in parallel and the tasks won't run long enough to block the worker from processing other waiting tasks.

However, executing a task does have overhead. A message needs to be sent, data may not be local, etc. So if the tasks are too fine-grained the additional overhead may not be worth it in the end.

See also:

The book Art of Concurrency has a whole section dedicated to the topic of task granularity.

Data locality

The worker processing the task should be as close to the data as possible. The best would be to have a copy in memory, the worst would be a full transfer from another continent.

If the data is far away, you could try to run another worker at location, or if that's not possible - cache often used data, or preload data you know is going to be used.

The easiest way to share data between workers is to use a distributed cache system, like memcached.

See also:

The paper Distributed Computing Economics by Jim Gray is an excellent introduction to the topic of data locality.

State

Since celery is a distributed system, you can't know in which process, or on what machine the task will be executed. You can't even know if the task will run in a timely manner.

The ancient async sayings tells us that "asserting the world is the responsibility of the task". What this means is that the world view may have changed since the task was requested, so the task is responsible for making sure the world is how it should be; If you have a task that re-indexes a search engine, and the search engine should only be re-indexed at maximum every 5 minutes, then it must be the tasks responsibility to assert that, not the callers.

Another gotcha is Django model objects. They shouldn't be passed on as arguments to tasks. It's almost always better to re-fetch the object from the database when the task is running instead, as using old data may lead to race conditions.

Imagine the following scenario where you have an article and a task that automatically expands some abbreviations in it:

```
class Article(models.Model):
    title = models.CharField()
    body = models.TextField()

@task
def expand_abbreviations(article):
    article.body.replace("MyCorp", "My Corporation")
    article.save()
```

First, an author creates an article and saves it, then the author clicks on a button that initiates the abbreviation task.

```
>>> article = Article.objects.get(id=102)
>>> expand_abbreviations.delay(model_object)
```

Now, the queue is very busy, so the task won't be run for another 2 minutes. In the meantime another author makes changes to the article, so when the task is finally run, the body of the article is reverted to the old version because the task had the old body in its argument.

Fixing the race condition is easy, just use the article id instead, and re-fetch the article in the task body:

```
@task
def expand_abbreviations(article_id):
    article = Article.objects.get(id=article_id)
    article.body.replace("MyCorp", "My Corporation")
    article.save()

>>> expand_abbreviations(article_id)
```

There might even be performance benefits to this approach, as sending large messages may be expensive.

Database transactions

Let's have a look at another example:

```
from django.db import transaction

@transaction.commit_on_success
def create_article(request):
    article = Article.objects.create(....)
    expand_abbreviations.delay(article.pk)
```

This is a Django view creating an article object in the database, then passing the primary key to a task. It uses the *commit_on_success* decorator, which will commit the transaction when the view returns, or roll back if the view raises an exception.

There is a race condition if the task starts executing before the transaction has been committed; The database object does not exist yet!

The solution is to *always commit transactions before sending tasks depending on state from the current transaction*:

```
@transaction.commit_manually
def create_article(request):
    try:
        article = Article.objects.create(...)
    except:
        transaction.rollback()
        raise
    else:
        transaction.commit()
        expand_abbreviations.delay(article.pk)
```

2.2.13 Example

Let's take a real wold example; A blog where comments posted needs to be filtered for spam. When the comment is created, the spam filter runs in the background, so the user doesn't have to wait for it to finish.

We have a Django blog application allowing comments on blog posts. We'll describe parts of the models/views and tasks for this application.

blog/models.py

The comment model looks like this:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class Comment(models.Model):
    name = models.CharField(_("name"), max_length=64)
    email_address = models.EmailField(_("email address"))
    homepage = models.URLField(_("home page"),
                               blank=True, verify_exists=False)
    comment = models.TextField(_("comment"))
    pub_date = models.DateTimeField(_("Published date"),
                                    editable=False, auto_add_now=True)
    is_spam = models.BooleanField(_("spam?"),
                                  default=False, editable=False)

    class Meta:
        verbose_name = _("comment")
        verbose_name_plural = _("comments")
```

In the view where the comment is posted, we first write the comment to the database, then we launch the spam filter task in the background.

blog/views.py

```
from django import forms
from django.http import HttpResponseRedirect
from django.template.context import RequestContext
from django.shortcuts import get_object_or_404, render_to_response

from blog import tasks
from blog.models import Comment


class CommentForm(forms.ModelForm):

    class Meta:
        model = Comment


def add_comment(request, slug, template_name="comments/create.html"):
    post = get_object_or_404(Entry, slug=slug)
    remote_addr = request.META.get("REMOTE_ADDR")

    if request.method == "post":
        form = CommentForm(request.POST, request.FILES)
        if form.is_valid():
            comment = form.save()
            # Check spam asynchronously.
            tasks.spam_filter.delay(comment_id=comment.id,
                                    remote_addr=remote_addr)
            return HttpResponseRedirect(post.get_absolute_url())
    else:
        form = CommentForm()
```

```
    context = RequestContext(request, {"form": form})
    return render_to_response(template_name, context_instance=context)
```

To filter spam in comments we use Akismet, the service used to filter spam in comments posted to the free weblog platform *Wordpress*. Akismet is free for personal use, but for commercial use you need to pay. You have to sign up to their service to get an API key.

To make API calls to Akismet we use the akismet.py library written by Michael Foord.

blog/tasks.py

```
from akismet import Akismet
from celery.task import task

from django.core.exceptions import ImproperlyConfigured
from django.contrib.sites.models import Site

from blog.models import Comment


@task
def spam_filter(comment_id, remote_addr=None):
    logger = spam_filter.get_logger()
    logger.info("Running spam filter for comment %s" % comment_id)

    comment = Comment.objects.get(pk=comment_id)
    current_domain = Site.objects.get_current().domain
    akismet = Akismet(settings.AKISMET_KEY, "http://%s" % domain)
    if not akismet.verify_key():
        raise ImproperlyConfigured("Invalid AKISMET_KEY")


    is_spam = akismet.comment_check(user_ip=remote_addr,
                        comment_content=comment.comment,
                        comment_author=comment.name,
                        comment_author_email=comment.email_address)
    if is_spam:
        comment.is_spam = True
        comment.save()

    return is_spam
```

2.3 Executing Tasks

- Basics
- ETA and countdown
- Expiration
- Serializers
- Connections and connection timeouts.
- Routing options
- AMQP options

2.3.1 Basics

Executing a task is done with `apply_async()`, and the shortcut: `delay()`.

delay is simple and convenient, as it looks like calling a regular function:

```
Task.delay(arg1, arg2, kwarg1="x", kwarg2="y")
```

The same using *apply_async* is written like this:

```
Task.apply_async(args=[arg1, arg2], kwargs={"kwarg1": "x", "kwarg2": "y"})
```

While *delay* is convenient, it doesn't give you as much control as using *apply_async*. With *apply_async* you can override the execution options available as attributes on the *Task* class (see *Task options*). In addition you can set countdown/eta, task expiry, provide a custom broker connection and more.

Let's go over these in more detail. All the examples uses a simple task called *add*, returning the sum of two positional arguments:

```
@task
def add(x, y):
    return x + y
```

Note: You can also execute a task by name using `send_task()`, if you don't have access to the task class:

```
>>> from celery.execute import send_task
>>> result = send_task("tasks.add", [2, 2])
>>> result.get()
4
```

2.3.2 ETA and countdown

The ETA (estimated time of arrival) lets you set a specific date and time that is the earliest time at which your task will be executed. *countdown* is a shortcut to set eta by seconds into the future.

```
>>> result = add.apply_async(args=[10, 10], countdown=3)
>>> result.get()    # this takes at least 3 seconds to return
20
```

The task is guaranteed to be executed at some time *after* the specified date and time, but not necessarily at that exact time. Possible reasons for broken deadlines may include many items waiting in the queue, or heavy network latency. To make sure your tasks are executed in a timely manner you should monitor queue lengths. Use Munin, or similar tools, to receive alerts, so appropriate action can be taken to ease the workload. See *Munin*.

While *countdown* is an integer, *eta* must be a `datetime` object, specifying an exact date and time (including millisecond precision, and timezone information):

```
>>> from datetime import datetime, timedelta

>>> tomorrow = datetime.now() + timedelta(days=1)
>>> add.apply_async(args=[10, 10], eta=tomorrow)
```

2.3.3 Expiration

The *expires* argument defines an optional expiry time, either as seconds after task publish, or a specific date and time using `datetime`:

```
>>> # Task expires after one minute from now.
>>> add.apply_async(args=[10, 10], expires=60)

>>> # Also supports datetime
>>> from datetime import datetime, timedelta
>>> add.apply_async(args=[10, 10], kwargs,
...                 expires=datetime.now() + timedelta(days=1)
```

When a worker receives an expired task it will mark the task as `REVOKED` (`TaskRevokedError`).

2.3.4 Serializers

Data transferred between clients and workers needs to be serialized. The default serializer is `pickle`, but you can change this globally or for each individual task. There is built-in support for `pickle`, *JSON*, *YAML* and *msgpack*, and you can also add your own custom serializers by registering them into the Kombu serializer registry (see Kombu: Serialization of Data).

Each option has its advantages and disadvantages.

json – JSON is supported in many programming languages, is now a standard part of Python (since 2.6), and is fairly fast to decode using the modern Python libraries such as `cjson` or `simplejson`.

The primary disadvantage to JSON is that it limits you to the following data types: strings, Unicode, floats, boolean, dictionaries, and lists. Decimals and dates are notably missing.

Also, binary data will be transferred using Base64 encoding, which will cause the transferred data to be around 34% larger than an encoding which supports native binary types.

However, if your data fits inside the above constraints and you need cross-language support, the default setting of JSON is probably your best choice.

See http://json.org for more information.

pickle – If you have no desire to support any language other than Python, then using the pickle encoding will gain you the support of all built-in Python data types (except class instances), smaller messages when sending binary files, and a slight speedup over JSON processing.

See http://docs.python.org/library/pickle.html for more information.

yaml – YAML has many of the same characteristics as json, except that it natively supports more data types (including dates, recursive references, etc.)

However, the Python libraries for YAML are a good bit slower than the libraries for JSON.

If you need a more expressive set of data types and need to maintain cross-language compatibility, then YAML may be a better fit than the above.

See http://yaml.org/ for more information.

msgpack – msgpack is a binary serialization format that is closer to JSON in features. It is very young however, and support should be considered experimental at this point.

See http://msgpack.org/ for more information.

The encoding used is available as a message header, so the worker knows how to deserialize any task. If you use a custom serializer, this serializer must be available for the worker.

The client uses the following order to decide which serializer to use when sending a task:

1. The *serializer* argument to *apply_async*
2. The tasks *serializer* attribute

3. The default CELERY_TASK_SERIALIZER setting.

- Using the *serializer* argument to *apply_async*:

```
>>> add.apply_async(args=[10, 10], serializer="json")
```

2.3.5 Connections and connection timeouts.

Automatic Pool Support

In version 2.3 there is now support for automatic connection pools, so you don't have to manually handle connections and publishers to reuse connections.

See the BROKER_POOL_LIMIT setting. This setting will be enabled by default in version 3.0.

You can handle the connection manually by creating a publisher:

```
numbers = [(2, 2), (4, 4), (8, 8), (16, 16)]

results = []
publisher = add.get_publisher()
try:
    for args in numbers:
        res = add.apply_async(args=args, publisher=publisher)
        results.append(res)
finally:
    publisher.close()
    publisher.connection.close()

print([res.get() for res in results])
```

Note: This particular example is better expressed as a task set. See *Task Sets*. Tasksets already reuses connections.

The connection timeout is the number of seconds to wait before giving up on establishing the connection. You can set this by using the *connect_timeout* argument to *apply_async*:

```
add.apply_async([10, 10], connect_timeout=3)
```

Or if you handle the connection manually:

```
publisher = add.get_publisher(connect_timeout=3)
```

2.3.6 Routing options

Celery uses the AMQP routing mechanisms to route tasks to different workers.

Messages (tasks) are sent to exchanges, a queue binds to an exchange with a routing key. Let's look at an example:

Let's pretend we have an application with lot of different tasks: some process video, others process images, and some gather collective intelligence about its users. Some of these tasks are more important, so we want to make sure the high priority tasks get sent to dedicated nodes.

For the sake of this example we have a single exchange called *tasks*. There are different types of exchanges, each type interpreting the routing key in different ways, implementing different messaging scenarios.

The most common types used with Celery are *direct* and *topic*.

- direct

  Matches the routing key exactly.

- topic

  In the topic exchange the routing key is made up of words separated by dots (.). Words can be matched by the wild cards * and #, where * matches one exact word, and # matches one or many words.

  For example, **.stock.#* matches the routing keys *usd.stock* and *euro.stock.db* but not *stock.nasdaq*.

We create three queues, *video*, *image* and *lowpri* that binds to the *tasks* exchange. For the queues we use the following binding keys:

```
video: video.#
image: image.#
lowpri: misc.#
```

Now we can send our tasks to different worker machines, by making the workers listen to different queues:

```
>>> add.apply_async(args=[filename],
...                                routing_key="video.compress")

>>> add.apply_async(args=[filename, 360],
...                              routing_key="image.rotate")

>>> add.apply_async(args=[filename, selection],
...                             routing_key="image.crop")
>>> add.apply_async(routing_key="misc.recommend")
```

Later, if the crop task is consuming a lot of resources, we can bind new workers to handle just the *"image.crop"* task, by creating a new queue that binds to *"image.crop*".

See also:

To find out more about routing, please see *Routing Tasks*.

2.3.7 AMQP options

- mandatory

This sets the delivery to be mandatory. An exception will be raised if there are no running workers able to take on the task.

Not supported by `amqplib`.

- immediate

Request immediate delivery. Will raise an exception if the task cannot be routed to a worker immediately.

Not supported by `amqplib`.

- priority

A number between *0* and *9*, where *0* is the highest priority.

Note: RabbitMQ does not yet support AMQP priorities.

2.4 Workers Guide

- Starting the worker
- Stopping the worker
- Restarting the worker
- Concurrency
- Persistent revokes
- Time limits
  - Changing time limits at runtime
- Max tasks per child setting
- Remote control
  - The `broadcast()` function.
  - Rate limits
  - Revoking tasks
  - Remote shutdown
  - Ping
  - Enable/disable events
  - Writing your own remote control commands
- Inspecting workers
  - Dump of registered tasks
  - Dump of currently executing tasks
  - Dump of scheduled (ETA) tasks
  - Dump of reserved tasks

2.4.1 Starting the worker

You can start celeryd to run in the foreground by executing the command:

```
$ celeryd --loglevel=INFO
```

You probably want to use a daemonization tool to start *celeryd* in the background. See *Running celeryd as a daemon* for help using *celeryd* with popular daemonization tools.

For a full list of available command line options see `celeryd`, or simply do:

```
$ celeryd --help
```

You can also start multiple workers on the same machine. If you do so be sure to give a unique name to each individual worker by specifying a host name with the *`--hostname|-n`* argument:

```
$ celeryd --loglevel=INFO --concurrency=10 -n worker1.example.com
$ celeryd --loglevel=INFO --concurrency=10 -n worker2.example.com
$ celeryd --loglevel=INFO --concurrency=10 -n worker3.example.com
```

2.4.2 Stopping the worker

Shutdown should be accomplished using the `TERM` signal.

When shutdown is initiated the worker will finish all currently executing tasks before it actually terminates, so if these tasks are important you should wait for it to finish before doing anything drastic (like sending the `KILL` signal).

If the worker won't shutdown after considerate time, for example because of tasks stuck in an infinite-loop, you can use the `KILL` signal to force terminate the worker, but be aware that currently executing tasks will be lost (unless the tasks have the `acks_late` option set).

Also as processes can't override the `KILL` signal, the worker will not be able to reap its children, so make sure to do so manually. This command usually does the trick:

```
$ ps auxww | grep celeryd | awk '{print $2}' | xargs kill -9
```

2.4.3 Restarting the worker

Other than stopping then starting the worker to restart, you can also restart the worker using the `HUP` signal:

```
$ kill -HUP $pid
```

The worker will then replace itself with a new instance using the same arguments as it was started with.

2.4.4 Concurrency

By default multiprocessing is used to perform concurrent execution of tasks, but you can also use *Eventlet*. The number of worker processes/threads can be changed using the `--concurrency` argument and defaults to the number of CPUs available on the machine.

Number of processes (multiprocessing)

More worker processes are usually better, but there's a cut-off point where adding more processes affects performance in negative ways. There is even some evidence to support that having multiple celeryd's running, may perform better than having a single worker. For example 3 celeryd's with 10 worker processes each. You need to experiment to find the numbers that works best for you, as this varies based on application, work load, task run times and other factors.

2.4.5 Persistent revokes

Revoking tasks works by sending a broadcast message to all the workers, the workers then keep a list of revoked tasks in memory.

If you want tasks to remain revoked after worker restart you need to specify a file for these to be stored in, either by using the *–statedb* argument to `celeryd` or the `CELERYD_STATE_DB` setting. See `CELERYD_STATE_DB` for more information.

2.4.6 Time limits

New in version 2.0.

A single task can potentially run forever, if you have lots of tasks waiting for some event that will never happen you will block the worker from processing new tasks indefinitely. The best way to defend against this scenario happening is enabling time limits.

The time limit (*–time-limit*) is the maximum number of seconds a task may run before the process executing it is terminated and replaced by a new process. You can also enable a soft time limit (*–soft-time-limit*), this raises an exception the task can catch to clean up before the hard time limit kills it:

```
from celery.task import task
from celery.exceptions import SoftTimeLimitExceeded

@task()
def mytask():
    try:
        do_work()
    except SoftTimeLimitExceeded:
        clean_up_in_a_hurry()
```

Time limits can also be set using the CELERYD_TASK_TIME_LIMIT / CELERYD_SOFT_TASK_TIME_LIMIT settings.

Note: Time limits do not currently work on Windows and other platforms that do not support the SIGUSR1 signal.

Changing time limits at runtime

New in version 2.3.

You can change the soft and hard time limits for a task by using the time_limit remote control command.

Example changing the time limit for the tasks.crawl_the_web task to have a soft time limit of one minute, and a hard time limit of two minutes:

```
>>> from celery.task import control
>>> control.time_limit("tasks.crawl_the_web",
                       soft=60, hard=120, reply=True)
[{'worker1.example.com': {'ok': 'time limits set successfully'}}]
```

Only tasks that starts executing after the time limit change will be affected.

2.4.7 Max tasks per child setting

With this option you can configure the maximum number of tasks a worker can execute before it's replaced by a new process.

This is useful if you have memory leaks you have no control over for example from closed source C extensions.

The option can be set using the *–maxtasksperchild* argument to *celeryd* or using the CELERYD_MAX_TASKS_PER_CHILD setting.

2.4.8 Remote control

New in version 2.0.

Workers have the ability to be remote controlled using a high-priority broadcast message queue. The commands can be directed to all, or a specific list of workers.

Commands can also have replies. The client can then wait for and collect those replies. Since there's no central authority to know how many workers are available in the cluster, there is also no way to estimate how many workers may send a reply, so the client has a configurable timeout — the deadline in seconds for replies to arrive in. This timeout defaults to one second. If the worker doesn't reply within the deadline it doesn't necessarily mean the worker didn't reply, or worse is dead, but may simply be caused by network latency or the worker being slow at processing commands, so adjust the timeout accordingly.

In addition to timeouts, the client can specify the maximum number of replies to wait for. If a destination is specified, this limit is set to the number of destination hosts.

See also:

The **celeryctl** program is used to execute remote control commands from the command line. It supports all of the commands listed below. See *celeryctl: Management Utility* for more information.

The `broadcast()` function.

This is the client function used to send commands to the workers. Some remote control commands also have higher-level interfaces using `broadcast()` in the background, like `rate_limit()` and `ping()`.

Sending the `rate_limit` command and keyword arguments:

```
>>> from celery.task.control import broadcast
>>> broadcast("rate_limit", arguments={"task_name": "myapp.mytask",
...                                    "rate_limit": "200/m"})
```

This will send the command asynchronously, without waiting for a reply. To request a reply you have to use the *reply* argument:

```
>>> broadcast("rate_limit", {"task_name": "myapp.mytask",
...                          "rate_limit": "200/m"}, reply=True)
[{'worker1.example.com': 'New rate limit set successfully'},
 {'worker2.example.com': 'New rate limit set successfully'},
 {'worker3.example.com': 'New rate limit set successfully'}]
```

Using the *destination* argument you can specify a list of workers to receive the command:

```
>>> broadcast
>>> broadcast("rate_limit", {"task_name": "myapp.mytask",
...                          "rate_limit": "200/m"}, reply=True,
...           destination=["worker1.example.com"])
[{'worker1.example.com': 'New rate limit set successfully'}]
```

Of course, using the higher-level interface to set rate limits is much more convenient, but there are commands that can only be requested using `broadcast()`.

Rate limits

Example changing the rate limit for the *myapp.mytask* task to accept 200 tasks a minute on all servers:

```
>>> from celery.task.control import rate_limit
>>> rate_limit("myapp.mytask", "200/m")
```

Example changing the rate limit on a single host by specifying the destination host name:

```
>>> rate_limit("myapp.mytask", "200/m",
...            destination=["worker1.example.com"])
```

Warning: This won't affect workers with the `CELERY_DISABLE_RATE_LIMITS` setting on. To re-enable rate limits then you have to restart the worker.

Revoking tasks

All worker nodes keeps a memory of revoked task ids, either in-memory or persistent on disk (see *Persistent revokes*).

When a worker receives a revoke request it will skip executing the task, but it won't terminate an already executing task unless the *terminate* option is set.

If *terminate* is set the worker child process processing the task will be terminated. The default signal sent is *TERM*, but you can specify this using the *signal* argument. Signal can be the uppercase name of any signal defined in the `signal` module in the Python Standard Library.

Terminating a task also revokes it.

Example

```
>>> from celery.task.control import revoke
>>> revoke("d9078da5-9915-40a0-bfa1-392c7bde42ed")

>>> revoke("d9078da5-9915-40a0-bfa1-392c7bde42ed",
...        terminate=True)

>>> revoke("d9078da5-9915-40a0-bfa1-392c7bde42ed",
...        terminate=True, signal="SIGKILL")
```

Remote shutdown

This command will gracefully shut down the worker remotely:

```
>>> broadcast("shutdown") # shutdown all workers
>>> broadcast("shutdown, destination="worker1.example.com")
```

Ping

This command requests a ping from alive workers. The workers reply with the string 'pong', and that's just about it. It will use the default one second timeout for replies unless you specify a custom timeout:

```
>>> from celery.task.control import ping
>>> ping(timeout=0.5)
[{'worker1.example.com': 'pong'},
 {'worker2.example.com': 'pong'},
 {'worker3.example.com': 'pong'}]
```

`ping()` also supports the *destination* argument, so you can specify which workers to ping:

```
>>> ping(['worker2.example.com', 'worker3.example.com'])
[{'worker2.example.com': 'pong'},
 {'worker3.example.com': 'pong'}]
```

Enable/disable events

You can enable/disable events by using the *enable_events*, *disable_events* commands. This is useful to temporarily monitor a worker using **celeryev/celerymon**.

```
>>> broadcast("enable_events")
>>> broadcast("disable_events")
```

Writing your own remote control commands

Remote control commands are registered in the control panel and they take a single argument: the current `ControlDispatch` instance. From there you have access to the active `Consumer` if needed.

Here's an example control command that restarts the broker connection:

```
from celery.worker.control import Panel

@Panel.register
def reset_connection(panel):
    panel.logger.critical("Connection reset by remote control.")
    panel.consumer.reset_connection()
    return {"ok": "connection reset"}
```

These can be added to task modules, or you can keep them in their own module then import them using the `CELERY_IMPORTS` setting:

```
CELERY_IMPORTS = ("myapp.worker.control", )
```

2.4.9 Inspecting workers

`celery.task.control.inspect` lets you inspect running workers. It uses remote control commands under the hood.

```
>>> from celery.task.control import inspect

# Inspect all nodes.
>>> i = inspect()

# Specify multiple nodes to inspect.
>>> i = inspect(["worker1.example.com", "worker2.example.com"])

# Specify a single node to inspect.
>>> i = inspect("worker1.example.com")
```

Dump of registered tasks

You can get a list of tasks registered in the worker using the `registered()`:

```
>>> i.registered()
[{'worker1.example.com': ['celery.delete_expired_task_meta',
                          'celery.execute_remote',
                          'celery.map_async',
                          'celery.ping',
                          'celery.task.http.HttpDispatchTask',
                          'tasks.add',
                          'tasks.sleeptask']}]
```

Dump of currently executing tasks

You can get a list of active tasks using `active()`:

```
>>> i.active()
[{'worker1.example.com':
    [{"name": "tasks.sleeptask",
      "id": "32666e9b-809c-41fa-8e93-5ae0c80afbbf",
      "args": "(8,)",
      "kwargs": "{}"}]}]
```

Dump of scheduled (ETA) tasks

You can get a list of tasks waiting to be scheduled by using `scheduled()`:

```
>>> i.scheduled()
[{'worker1.example.com':
    [{"eta": "2010-06-07 09:07:52", "priority": 0,
      "request": {
        "name": "tasks.sleeptask",
        "id": "1a7980ea-8b19-413e-91d2-0b74f3844c4d",
        "args": "[1]",
        "kwargs": "{}"}},
     {"eta": "2010-06-07 09:07:53", "priority": 0,
      "request": {
        "name": "tasks.sleeptask",
        "id": "49661b9a-aa22-4120-94b7-9ee8031d219d",
        "args": "[2]",
        "kwargs": "{}"}}]}]
```

Note that these are tasks with an eta/countdown argument, not periodic tasks.

Dump of reserved tasks

Reserved tasks are tasks that has been received, but is still waiting to be executed.

You can get a list of these using `reserved()`:

```
>>> i.reserved()
[{'worker1.example.com':
    [{"name": "tasks.sleeptask",
      "id": "32666e9b-809c-41fa-8e93-5ae0c80afbbf",
      "args": "(8,)",
      "kwargs": "{}"}]}]
```

2.5 Periodic Tasks

- Introduction
- Entries
  - Available Fields
- Crontab schedules
- Starting celerybeat
  - Using custom scheduler classes

2.5.1 Introduction

celerybeat is a scheduler. It kicks off tasks at regular intervals, which are then executed by the worker nodes available in the cluster.

By default the entries are taken from the `CELERYBEAT_SCHEDULE` setting, but custom stores can also be used, like storing the entries in an SQL database.

You have to ensure only a single scheduler is running for a schedule at a time, otherwise you would end up with duplicate tasks. Using a centralized approach means the schedule does not have to be synchronized, and the service can operate without using locks.

2.5.2 Entries

To schedule a task periodically you have to add an entry to the `CELERYBEAT_SCHEDULE` setting.

Example: Run the *tasks.add* task every 30 seconds.

```
from datetime import timedelta

CELERYBEAT_SCHEDULE = {
    "runs-every-30-seconds": {
        "task": "tasks.add",
        "schedule": timedelta(seconds=30),
        "args": (16, 16)
    },
}
```

Using a `timedelta` for the schedule means the task will be executed 30 seconds after *celerybeat* starts, and then every 30 seconds after the last run. A crontab like schedule also exists, see the section on Crontab schedules.

Available Fields

- *task*

  The name of the task to execute.

- *schedule*

  The frequency of execution.

  This can be the number of seconds as an integer, a `timedelta`, or a `crontab`. You can also define your own custom schedule types, by extending the interface of `schedule`.

- *args*

  Positional arguments (`list` or `tuple`).

- *kwargs*

  Keyword arguments (`dict`).

- *options*

  Execution options (`dict`).

  This can be any argument supported by `apply_async()`, e.g. *exchange*, *routing_key*, *expires*, and so on.

- *relative*

By default `timedelta` schedules are scheduled "by the clock". This means the frequency is rounded to the nearest second, minute, hour or day depending on the period of the timedelta.

If *relative* is true the frequency is not rounded and will be relative to the time when **celerybeat** was started.

2.5.3 Crontab schedules

If you want more control over when the task is executed, for example, a particular time of day or day of the week, you can use the *crontab* schedule type:

```
from celery.schedules import crontab

CELERYBEAT_SCHEDULE = {
    # Executes every Monday morning at 7:30 A.M
    "every-monday-morning": {
        "task": "tasks.add",
        "schedule": crontab(hour=7, minute=30, day_of_week=1),
        "args": (16, 16),
    },
}
```

The syntax of these crontab expressions are very flexible. Some examples:

Example	Meaning
`crontab()`	Execute every minute.
`crontab(minute=0, hour=0)`	Execute daily at midnight.
`crontab(minute=0, hour="*/3")`	Execute every three hours: 3am, 6am, 9am, noon, 3pm, 6pm, 9pm.
`crontab(minute=0,` `hour=[0,3,6,9,12,15,18,21])`	Same as previous.
`crontab(minute="*/15")`	Execute every 15 minutes.
`crontab(day_of_week="sunday")`	Execute every minute (!) at Sundays.
`crontab(minute="*",` `hour="*", day_of_week="sun")`	Same as previous.
`crontab(minute="*/10",` `hour="3,17,22", day_of_week="thu,fri")`	Execute every ten minutes, but only between 3-4 am, 5-6 pm and 10-11 pm on Thursdays or Fridays.
`crontab(minute=0, hour="*/2,*/3")`	Execute every even hour, and every hour divisible by three. This means: at every hour *except*: 1am, 5am, 7am, 11am, 1pm, 5pm, 7pm, 11pm
`crontab(minute=0, hour="*/5")`	Execute hour divisible by 5. This means that it is triggered at 3pm, not 5pm (since 3pm equals the 24-hour clock value of "15", which is divisible by 5).
`crontab(minute=0, hour="*/3,8-17")`	Execute every hour divisible by 3, and every hour during office hours (8am-5pm).

2.5.4 Starting celerybeat

To start the **celerybeat** service:

```
$ celerybeat
```

You can also start *celerybeat* with *celeryd* by using the *-B* option, this is convenient if you only intend to use one worker node:

```
$ celeryd -B
```

Celerybeat needs to store the last run times of the tasks in a local database file (named *celerybeat-schedule* by default), so it needs access to write in the current directory, or alternatively you can specify a custom location for this file:

```
$ celerybeat -s /home/celery/var/run/celerybeat-schedule
```

Note: To daemonize celerybeat see *Running celeryd as a daemon*.

Using custom scheduler classes

Custom scheduler classes can be specified on the command line (the *-S* argument). The default scheduler is `celery.beat.PersistentScheduler`, which is simply keeping track of the last run times in a local database file (a `shelve`).

django-celery also ships with a scheduler that stores the schedule in the Django database:

```
$ celerybeat -S djcelery.schedulers.DatabaseScheduler
```

Using *django-celery*'s scheduler you can add, modify and remove periodic tasks from the Django Admin.

2.6 Sets of tasks, Subtasks and Callbacks

- Subtasks
  - Callbacks
- Task Sets
  - Results
- Chords
  - Important Notes

2.6.1 Subtasks

New in version 2.0.

The `subtask` type is used to wrap the arguments and execution options for a single task invocation:

```
subtask(task_name_or_cls, args, kwargs, options)
```

For convenience every task also has a shortcut to create subtasks:

```
task.subtask(args, kwargs, options)
```

`subtask` is actually a `dict` subclass, which means it can be serialized with JSON or other encodings that doesn't support complex Python objects.

Also it can be regarded as a type, as the following usage works:

```
>>> s = subtask("tasks.add", args=(2, 2), kwargs={})

>>> subtask(dict(s))  # coerce dict into subtask
```

This makes it excellent as a means to pass callbacks around to tasks.

Callbacks

Let's improve our *add* task so it can accept a callback that takes the result as an argument:

```
from celery.task import task
from celery.task.sets import subtask

@task
def add(x, y, callback=None):
    result = x + y
    if callback is not None:
        subtask(callback).delay(result)
    return result
```

`subtask` also knows how it should be applied, asynchronously by `delay()`, and eagerly by `apply()`.

The best thing is that any arguments you add to *subtask.delay*, will be prepended to the arguments specified by the subtask itself!

If you have the subtask:

```
>>> add.subtask(args=(10, ))
```

subtask.delay(result) becomes:

```
>>> add.apply_async(args=(result, 10))
```

...

Now let's execute our new *add* task with a callback:

```
>>> add.delay(2, 2, callback=add.subtask((8, )))
```

As expected this will first launch one task calculating $2 + 2$, then another task calculating $4 + 8$.

2.6.2 Task Sets

The `TaskSet` enables easy invocation of several tasks at once, and is then able to join the results in the same order as the tasks were invoked.

A task set takes a list of `subtask`'s:

```
>>> from celery.task.sets import TaskSet
>>> from tasks import add

>>> job = TaskSet(tasks=[
...             add.subtask((4, 4)),
...             add.subtask((8, 8)),
...             add.subtask((16, 16)),
...             add.subtask((32, 32)),
... ])

>>> result = job.apply_async()
```

```
>>> result.ready()  # have all subtasks completed?
True
>>> result.successful() # were all subtasks successful?
True
>>> result.join()
[4, 8, 16, 32, 64]
```

Results

When a `TaskSet` is applied it returns a `TaskSetResult` object.

`TaskSetResult` takes a list of `AsyncResult` instances and operates on them as if it was a single task.

It supports the following operations:

- `successful()`

  Returns `True` if all of the subtasks finished successfully (e.g. did not raise an exception).

- `failed()`

  Returns `True` if any of the subtasks failed.

- `waiting()`

  Returns `True` if any of the subtasks is not ready yet.

- `ready()`

  Return `True` if all of the subtasks are ready.

- `completed_count()`

  Returns the number of completed subtasks.

- `revoke()`

  Revokes all of the subtasks.

- `iterate()`

  Iterates over the return values of the subtasks as they finish, one by one.

- `join()`

  Gather the results for all of the subtasks and return a list with them ordered by the order of which they were called.

2.6.3 Chords

New in version 2.3.

A chord is a task that only executes after all of the tasks in a taskset has finished executing.

Let's calculate the sum of the expression $1 + 1 + 2 + 2 + 3 + 3...n + n$ up to a hundred digits.

First we need two tasks, `add()` and `tsum()` (`sum()` is already a standard function):

```
from celery.task import task

@task
def add(x, y):
    return x + y
```

```
@task
def tsum(numbers):
    return sum(numbers)
```

Now we can use a chord to calculate each addition step in parallel, and then get the sum of the resulting numbers:

```
>>> from celery.task import chord
>>> from tasks import add, tsum

>>> chord(add.subtask((i, i))
...     for i in xrange(100))(tsum.subtask()).get()
9900
```

This is obviously a very contrived example, the overhead of messaging and synchronization makes this a lot slower than its Python counterpart:

```
sum(i + i for i in xrange(100))
```

The synchronization step is costly, so you should avoid using chords as much as possible. Still, the chord is a powerful primitive to have in your toolbox as synchronization is a required step for many parallel algorithms.

Let's break the chord expression down:

```
>>> callback = tsum.subtask()
>>> header = [add.subtask((i, i)) for i in xrange(100)]
>>> result = chord(header)(callback)
>>> result.get()
9900
```

Remember, the callback can only be executed after all of the tasks in the header has returned. Each step in the header is executed as a task, in parallel, possibly on different nodes. The callback is then applied with the return value of each task in the header. The task id returned by `chord()` is the id of the callback, so you can wait for it to complete and get the final return value (but remember to *never have a task wait for other tasks*)

Important Notes

By default the synchronization step is implemented by having a recurring task poll the completion of the taskset every second, applying the subtask when ready.

Example implementation:

```
def unlock_chord(taskset, callback, interval=1, max_retries=None):
    if taskset.ready():
        return subtask(callback).delay(taskset.join())
    unlock_chord.retry(countdown=interval, max_retries=max_retries)
```

This is used by all result backends except Redis, which increments a counter after each task in the header, then applying the callback when the counter exceeds the number of tasks in the set. *Note:* chords do not properly work with Redis before version 2.2; you will need to upgrade to at least 2.2 to use them.

The Redis approach is a much better solution, but not easily implemented in other backends (suggestions welcome!).

Note: If you are using chords with the Redis result backend and also overriding the `Task.after_return()` method, you need to make sure to call the super method or else the chord callback will not be applied.

```
def after_return(self, *args, **kwargs):
    do_something()
    super(MyTask, self).after_return(*args, **kwargs)
```

2.7 HTTP Callback Tasks (Webhooks)

- Basics
- Django webhook example
- Ruby on Rails webhook example
- Executing webhook tasks

2.7.1 Basics

If you need to call into another language, framework or similar, you can do so by using HTTP callback tasks.

The HTTP callback tasks uses GET/POST data to pass arguments and returns result as a JSON response. The scheme to call a task is:

```
GET http://example.com/mytask/?arg1=a&arg2=b&arg3=c
```

or using POST:

```
POST http://example.com/mytask
```

Note: POST data needs to be form encoded.

Whether to use GET or POST is up to you and your requirements.

The web page should then return a response in the following format if the execution was successful:

```
{"status": "success", "retval": ....}
```

or if there was an error:

```
{"status": "failure": "reason": "Invalid moon alignment."}
```

2.7.2 Django webhook example

With this information you could define a simple task in Django:

```
from django.http import HttpResponse
from anyjson import serialize


def multiply(request):
    x = int(request.GET["x"])
    y = int(request.GET["y"])
    result = x * y
    response = {"status": "success", "retval": result}
    return HttpResponse(serialize(response), mimetype="application/json")
```

2.7.3 Ruby on Rails webhook example

or in Ruby on Rails:

```
def multiply
    @x = params[:x].to_i
    @y = params[:y].to_i

    @status = {:status => "success", :retval => @x * @y}

    render :json => @status
end
```

You can easily port this scheme to any language/framework; new examples and libraries are very welcome.

2.7.4 Executing webhook tasks

To execute the task you use the `URL` class:

```
>>> from celery.task.http import URL
>>> res = URL("http://example.com/multiply").get_async(x=10, y=10)
```

`URL` is a shortcut to the `HttpDispatchTask`. You can subclass this to extend the functionality.

```
>>> from celery.task.http import HttpDispatchTask
>>> res = HttpDispatchTask.delay(url="http://example.com/multiply", method="GET", x=10, y=10)
>>> res.get()
100
```

The output of **celeryd** (or the log file if enabled) should show the task being executed:

```
[INFO/MainProcess] Task celery.task.http.HttpDispatchTask
        [f2cc8efc-2a14-40cd-85ad-f1c77c94beeb] processed: 100
```

Since applying tasks can be done via HTTP using the `djcelery.views.apply()` view, executing tasks from other languages is easy. For an example service exposing tasks via HTTP you should have a look at *examples/celery_http_gateway* in the Celery distribution: http://github.com/ask/celery/tree/master/examples/celery_http_gateway/

2.8 Routing Tasks

Warning: This document refers to functionality only available in brokers using AMQP. Other brokers may implement some functionality, see their respective documentation for more information, or contact the *Mailing list*.

- Basics
  - Automatic routing
    - Changing the name of the default queue
    - How the queues are defined
  - Manual routing
- AMQP Primer
  - Messages
  - Producers, consumers and brokers
  - Exchanges, queues and routing keys.
  - Exchange types
    - Direct exchanges
    - Topic exchanges
  - Related API commands
  - Hands-on with the API
- Routing Tasks
  - Defining queues
  - Specifying task destination
  - Routers

2.8.1 Basics

Automatic routing

The simplest way to do routing is to use the `CELERY_CREATE_MISSING_QUEUES` setting (on by default).

With this setting on, a named queue that is not already defined in `CELERY_QUEUES` will be created automatically. This makes it easy to perform simple routing tasks.

Say you have two servers, *x*, and *y* that handles regular tasks, and one server *z*, that only handles feed related tasks. You can use this configuration:

```
CELERY_ROUTES = {"feed.tasks.import_feed": {"queue": "feeds"}}
```

With this route enabled import feed tasks will be routed to the *"feeds"* queue, while all other tasks will be routed to the default queue (named *"celery"* for historical reasons).

Now you can start server *z* to only process the feeds queue like this:

```
(z)$ celeryd -Q feeds
```

You can specify as many queues as you want, so you can make this server process the default queue as well:

```
(z)$ celeryd -Q feeds,celery
```

Changing the name of the default queue

You can change the name of the default queue by using the following configuration:

```
CELERY_QUEUES = {"default": {"exchange": "default",
                             "binding_key": "default"}}
CELERY_DEFAULT_QUEUE = "default"
```

How the queues are defined

The point with this feature is to hide the complex AMQP protocol for users with only basic needs. However – you may still be interested in how these queues are declared.

A queue named *"video"* will be created with the following settings:

```
{"exchange": "video",
 "exchange_type": "direct",
 "routing_key": "video"}
```

The non-AMQP backends like *ghettoq* does not support exchanges, so they require the exchange to have the same name as the queue. Using this design ensures it will work for them as well.

Manual routing

Say you have two servers, *x*, and *y* that handles regular tasks, and one server *z*, that only handles feed related tasks, you can use this configuration:

```
CELERY_DEFAULT_QUEUE = "default"
CELERY_QUEUES = {
    "default": {
        "binding_key": "task.#",
    },
    "feed_tasks": {
        "binding_key": "feed.#",
    },
}
CELERY_DEFAULT_EXCHANGE = "tasks"
CELERY_DEFAULT_EXCHANGE_TYPE = "topic"
CELERY_DEFAULT_ROUTING_KEY = "task.default"
```

`CELERY_QUEUES` is a map of queue names and their exchange/type/binding_key, if you don't set exchange or exchange type, they will be taken from the `CELERY_DEFAULT_EXCHANGE` and `CELERY_DEFAULT_EXCHANGE_TYPE` settings.

To route a task to the *feed_tasks* queue, you can add an entry in the `CELERY_ROUTES` setting:

```
CELERY_ROUTES = {
        "feeds.tasks.import_feed": {
            "queue": "feed_tasks",
            "routing_key": "feed.import",
        },
}
```

You can also override this using the *routing_key* argument to `apply_async()`, or `send_task()`:

```
>>> from feeds.tasks import import_feed
>>> import_feed.apply_async(args=["http://cnn.com/rss"],
...                         queue="feed_tasks",
...                         routing_key="feed.import")
```

To make server *z* consume from the feed queue exclusively you can start it with the `-Q` option:

```
(z)$ celeryd -Q feed_tasks --hostname=z.example.com
```

Servers *x* and *y* must be configured to consume from the default queue:

```
(x)$ celeryd -Q default --hostname=x.example.com
(y)$ celeryd -Q default --hostname=y.example.com
```

If you want, you can even have your feed processing worker handle regular tasks as well, maybe in times when there's a lot of work to do:

```
(z)$ celeryd -Q feed_tasks,default --hostname=z.example.com
```

If you have another queue but on another exchange you want to add, just specify a custom exchange and exchange type:

```
CELERY_QUEUES = {
        "feed_tasks": {
            "binding_key": "feed.#",
        },
        "regular_tasks": {
            "binding_key": "task.#",
        },
        "image_tasks": {
            "binding_key": "image.compress",
            "exchange": "mediatasks",
            "exchange_type": "direct",
        },
    }
```

If you're confused about these terms, you should read up on AMQP.

See also:

In addition to the *AMQP Primer* below, there's Rabbits and Warrens, an excellent blog post describing queues and exchanges. There's also AMQP in 10 minutes*: Flexible Routing Model, and Standard Exchange Types. For users of RabbitMQ the RabbitMQ FAQ could be useful as a source of information.

2.8.2 AMQP Primer

Messages

A message consists of headers and a body. Celery uses headers to store the content type of the message and its content encoding. The content type is usually the serialization format used to serialize the message. The body contains the name of the task to execute, the task id (UUID), the arguments to execute it with and some additional metadata – like the number of retries or an ETA.

This is an example task message represented as a Python dictionary:

```
{"task": "myapp.tasks.add",
 "id": "54086c5e-6193-4575-8308-dbab76798756",
 "args": [4, 4],
 "kwargs": {}}
```

Producers, consumers and brokers

The client sending messages is typically called a *publisher*, or a *producer*, while the entity receiving messages is called a *consumer*.

The *broker* is the message server, routing messages from producers to consumers.

You are likely to see these terms used a lot in AMQP related material.

Exchanges, queues and routing keys.

1. Messages are sent to exchanges.
2. An exchange routes messages to one or more queues. Several exchange types exists, providing different ways to do routing, or implementing different messaging scenarios.
3. The message waits in the queue until someone consumes it.
4. The message is deleted from the queue when it has been acknowledged.

The steps required to send and receive messages are:

1. Create an exchange
2. Create a queue
3. Bind the queue to the exchange.

Celery automatically creates the entities necessary for the queues in `CELERY_QUEUES` to work (except if the queue's *auto_declare* setting is set to `False`).

Here's an example queue configuration with three queues; One for video, one for images and one default queue for everything else:

```
CELERY_QUEUES = {
    "default": {
        "exchange": "default",
        "binding_key": "default"},
    "videos": {
        "exchange": "media",
        "binding_key": "media.video",
    },
    "images": {
        "exchange": "media",
        "binding_key": "media.image",
    }
}
CELERY_DEFAULT_QUEUE = "default"
CELERY_DEFAULT_EXCHANGE_TYPE = "direct"
CELERY_DEFAULT_ROUTING_KEY = "default"
```

Note: In Celery the *routing_key* is the key used to send the message, while *binding_key* is the key the queue is bound with. In the AMQP API they are both referred to as the routing key.

Exchange types

The exchange type defines how the messages are routed through the exchange. The exchange types defined in the standard are *direct*, *topic*, *fanout* and *headers*. Also non-standard exchange types are available as plug-ins to RabbitMQ, like the last-value-cache plug-in by Michael Bridgen.

Direct exchanges

Direct exchanges match by exact routing keys, so a queue bound by the routing key *video* only receives messages with that routing key.

Topic exchanges

Topic exchanges matches routing keys using dot-separated words, and the wildcard characters: `*` (matches a single word), and # (matches zero or more words).

With routing keys like `usa.news`, `usa.weather`, `norway.news` and `norway.weather`, bindings could be `*.news` (all news), `usa.#` (all items in the USA) or `usa.weather` (all USA weather items).

Related API commands

`exchange.declare(exchange_name, type, passive, durable, auto_delete, internal)`
Declares an exchange by name.

Parameters

- **passive** – Passive means the exchange won't be created, but you can use this to check if the exchange already exists.
- **durable** – Durable exchanges are persistent. That is - they survive a broker restart.
- **auto_delete** – This means the queue will be deleted by the broker when there are no more queues using it.

`queue.`**`declare`**(*queue_name*, *passive*, *durable*, *exclusive*, *auto_delete*)
Declares a queue by name.

Exclusive queues can only be consumed from by the current connection. Exclusive also implies *auto_delete*.

`queue.`**`bind`**(*queue_name*, *exchange_name*, *routing_key*)
Binds a queue to an exchange with a routing key. Unbound queues will not receive messages, so this is necessary.

`queue.`**`delete`**(*name*, *if_unused=False*, *if_empty=False*)
Deletes a queue and its binding.

`exchange.`**`delete`**(*name*, *if_unused=False*)
Deletes an exchange.

Note: Declaring does not necessarily mean "create". When you declare you *assert* that the entity exists and that it's operable. There is no rule as to whom should initially create the exchange/queue/binding, whether consumer or producer. Usually the first one to need it will be the one to create it.

Hands-on with the API

Celery comes with a tool called **camqadm** (short for Celery AMQ Admin). It's used for command-line access to the AMQP API, enabling access to administration tasks like creating/deleting queues and exchanges, purging queues or sending messages.

You can write commands directly in the arguments to **camqadm**, or just start with no arguments to start it in shell-mode:

```
$ camqadm
-> connecting to amqp://guest@localhost:5672/.
-> connected.
1>
```

Here `1>` is the prompt. The number 1, is the number of commands you have executed so far. Type `help` for a list of commands available. It also supports auto-completion, so you can start typing a command and then hit the *tab* key to show a list of possible matches.

Let's create a queue we can send messages to:

```
1> exchange.declare testexchange direct
ok.
2> queue.declare testqueue
ok. queue:testqueue messages:0 consumers:0.
3> queue.bind testqueue testexchange testkey
ok.
```

This created the direct exchange `testexchange`, and a queue named `testqueue`. The queue is bound to the exchange using the routing key `testkey`.

From now on all messages sent to the exchange `testexchange` with routing key `testkey` will be moved to this queue. We can send a message by using the `basic.publish` command:

```
4> basic.publish "This is a message!" testexchange testkey
ok.
```

Now that the message is sent we can retrieve it again. We use the `basic.get` ` command here, which polls for new messages on the queue.

Pop a message off the queue:

```
5> basic.get testqueue
{'body': 'This is a message!',
 'delivery_info': {'delivery_tag': 1,
                   'exchange': u'testexchange',
                   'message_count': 0,
                   'redelivered': False,
                   'routing_key': u'testkey'},
 'properties': {}}
```

AMQP uses acknowledgment to signify that a message has been received and processed successfully. If the message has not been acknowledged and consumer channel is closed, the message will be delivered to another consumer.

Note the delivery tag listed in the structure above; Within a connection channel, every received message has a unique delivery tag, This tag is used to acknowledge the message. Also note that delivery tags are not unique across connections, so in another client the delivery tag *1* might point to a different message than in this channel.

You can acknowledge the message we received using `basic.ack`:

```
6> basic.ack 1
ok.
```

To clean up after our test session we should delete the entities we created:

```
7> queue.delete testqueue
ok. 0 messages deleted.
8> exchange.delete testexchange
ok.
```

2.8.3 Routing Tasks

Defining queues

In Celery available queues are defined by the `CELERY_QUEUES` setting.

Here's an example queue configuration with three queues; One for video, one for images and one default queue for everything else:

```
CELERY_QUEUES = {
    "default": {
        "exchange": "default",
        "binding_key": "default"},
    "videos": {
        "exchange": "media",
        "exchange_type": "topic",
        "binding_key": "media.video",
    },
    "images": {
        "exchange": "media",
        "exchange_type": "topic",
        "binding_key": "media.image",
    }
}
CELERY_DEFAULT_QUEUE = "default"
CELERY_DEFAULT_EXCHANGE = "default"
CELERY_DEFAULT_EXCHANGE_TYPE = "direct"
CELERY_DEFAULT_ROUTING_KEY = "default"
```

Here, the `CELERY_DEFAULT_QUEUE` will be used to route tasks that doesn't have an explicit route.

The default exchange, exchange type and routing key will be used as the default routing values for tasks, and as the default values for entries in `CELERY_QUEUES`.

Specifying task destination

The destination for a task is decided by the following (in order):

1. The *Routers* defined in `CELERY_ROUTES`.
2. The routing arguments to `apply_async()`.
3. Routing related attributes defined on the `Task` itself.

It is considered best practice to not hard-code these settings, but rather leave that as configuration options by using *Routers*; This is the most flexible approach, but sensible defaults can still be set as task attributes.

Routers

A router is a class that decides the routing options for a task.

All you need to define a new router is to create a class with a `route_for_task` method:

```
class MyRouter(object):

    def route_for_task(self, task, args=None, kwargs=None):
        if task == "myapp.tasks.compress_video":
            return {"exchange": "video",
                    "exchange_type": "topic",
                    "routing_key": "video.compress"}
        return None
```

If you return the `queue` key, it will expand with the defined settings of that queue in `CELERY_QUEUES`:

```
{"queue": "video", "routing_key": "video.compress"}

becomes -->

    {"queue": "video",
     "exchange": "video",
     "exchange_type": "topic",
     "routing_key": "video.compress"}
```

You install router classes by adding them to the CELERY_ROUTES setting:

```
CELERY_ROUTES = (MyRouter(), )
```

Router classes can also be added by name:

```
CELERY_ROUTES = ("myapp.routers.MyRouter", )
```

For simple task name -> route mappings like the router example above, you can simply drop a dict into CELERY_ROUTES to get the same behavior:

```
CELERY_ROUTES = ({"myapp.tasks.compress_video": {
                        "queue": "video",
                        "routing_key": "video.compress"
                  }}, )
```

The routers will then be traversed in order, it will stop at the first router returning a true value, and use that as the final route for the task.

2.9 Monitoring and Management Guide

- Introduction
- Workers
  - celeryctl: Management Utility
    - Commands
    - Specifying destination nodes
  - Django Admin Monitor
    - Starting the monitor
    - Shutter frequency
    - Using outside of Django
  - celeryev: Curses Monitor
  - celerymon: Web monitor
- RabbitMQ
  - Inspecting queues
- Munin
- Events
  - Snapshots
    - Custom Camera
  - Event Reference
    - Task Events
    - Worker Events

2.9.1 Introduction

There are several tools available to monitor and inspect Celery clusters.

This document describes some of these, as as well as features related to monitoring, like events and broadcast commands.

2.9.2 Workers

celeryctl: Management Utility

New in version 2.1.

`celeryctl` is a command line utility to inspect and manage worker nodes (and to some degree tasks).

To list all the commands available do:

```
$ celeryctl help
```

or to get help for a specific command do:

```
$ celeryctl <command> --help
```

Commands

- **status: List active nodes in this cluster**

  ```
  $ celeryctl status
  ```

- **result: Show the result of a task**

  ```
  $ celeryctl result -t tasks.add 4e196aa4-0141-4601-8138-7aa33db0f577
  ```

  Note that you can omit the name of the task as long as the task doesn't use a custom result backend.

- **purge: Purge messages from all configured task queues.**

  ```
  $ celeryctl purge
  ```

  Warning: There is no undo for this operation, and messages will be permanently deleted!

- **inspect active: List active tasks**

  ```
  $ celeryctl inspect active
  ```

  These are all the tasks that are currently being executed.

- **inspect scheduled: List scheduled ETA tasks**

  ```
  $ celeryctl inspect scheduled
  ```

  These are tasks reserved by the worker because they have the *eta* or *countdown* argument set.

- **inspect reserved: List reserved tasks**

  ```
  $ celeryctl inspect reserved
  ```

This will list all tasks that have been prefetched by the worker, and is currently waiting to be executed (does not include tasks with an eta).

- **inspect revoked: List history of revoked tasks**

```
$ celeryctl inspect revoked
```

- **inspect registered: List registered tasks**

```
$ celeryctl inspect registered
```

- **inspect stats: Show worker statistics**

```
$ celeryctl inspect stats
```

- **inspect enable_events: Enable events**

```
$ celeryctl inspect enable_events
```

- **inspect disable_events: Disable events**

```
$ celeryctl inspect disable_events
```

Note: All `inspect` commands supports a `--timeout` argument, This is the number of seconds to wait for responses. You may have to increase this timeout if you're not getting a response due to latency.

Specifying destination nodes

By default the inspect commands operates on all workers. You can specify a single, or a list of workers by using the *–destination* argument:

```
$ celeryctl inspect -d w1,w2 reserved
```

Django Admin Monitor

New in version 2.1.

When you add django-celery to your Django project you will automatically get a monitor section as part of the Django admin interface.

This can also be used if you're not using Celery with a Django project.

Screenshot

Starting the monitor

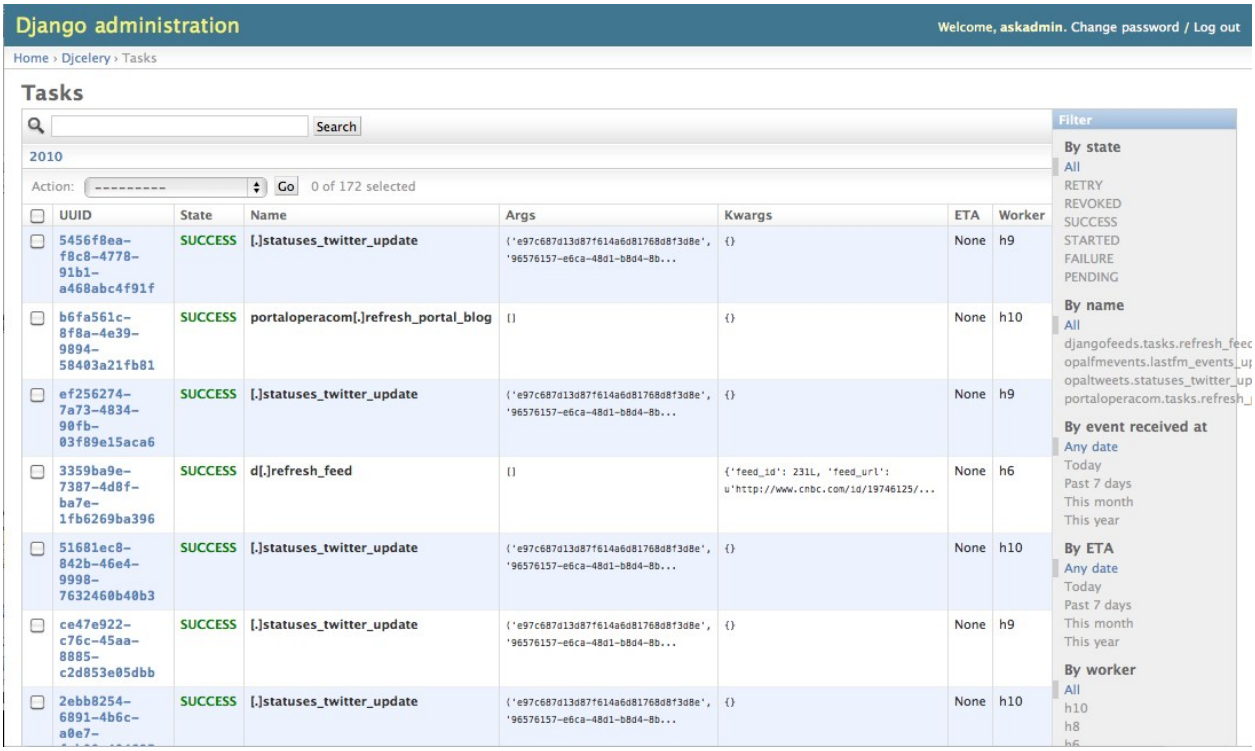
The Celery section will already be present in your admin interface, but you won't see any data appearing until you start the snapshot camera.

The camera takes snapshots of the events your workers sends at regular intervals, storing them in your database (See *Snapshots*).

To start the camera run:

```
$ python manage.py celerycam
```

If you haven't already enabled the sending of events you need to do so:




```
$ python manage.py celeryctl inspect enable_events
```

Tip You can enable events when the worker starts using the *-E* argument to `celeryd`.

Now that the camera has been started, and events have been enabled you should be able to see your workers and the tasks in the admin interface (it may take some time for workers to show up).

The admin interface shows tasks, worker nodes, and even lets you perform some actions, like revoking and rate limiting tasks, or shutting down worker nodes.

Shutter frequency

By default the camera takes a snapshot every second, if this is too frequent or you want to have higher precision, then you can change this using the `--frequency` argument. This is a float describing how often, in seconds, it should wake up to check if there are any new events:

```
$ python manage.py celerycam --frequency=3.0
```

The camera also supports rate limiting using the `--maxrate` argument. While the frequency controls how often the camera thread wakes up, the rate limit controls how often it will actually take a snapshot.

The rate limits can be specified in seconds, minutes or hours by appending */s*, */m* or */h* to the value. Example: `--maxrate=100/m`, means "hundred writes a minute".

The rate limit is off by default, which means it will take a snapshot for every `--frequency` seconds.

The events also expire after some time, so the database doesn't fill up. Successful tasks are deleted after 1 day, failed tasks after 3 days, and tasks in other states after 5 days.

Using outside of Django

django-celery also installs the **djcelerymon** program. This can be used by non-Django users, and runs both a web server and a snapshot camera in the same process.

Installing

Using **pip**:

```
$ pip install -U django-celery
```

or using **easy_install**:

```
$ easy_install -U django-celery
```

Running

djcelerymon reads configuration from your Celery configuration module, and sets up the Django environment using the same settings:

```
$ djcelerymon
```

Database tables will be created the first time the monitor is run. By default an *sqlite3* database file named `djcelerymon.db` is used, so make sure this file is writeable by the user running the monitor.

If you want to store the events in a different database, e.g. MySQL, then you can configure the *DATABASE** settings directly in your Celery config module. See http://docs.djangoproject.com/en/dev/ref/settings/#databases for more information about the database options available.

You will also be asked to create a superuser (and you need to create one to be able to log into the admin later):

```
Creating table auth_permission
Creating table auth_group_permissions
[...]

You just installed Django's auth system, which means you don't
have any superusers defined.  Would you like to create
one now? (yes/no): yes
Username (Leave blank to use 'username'): username
Email address: me@example.com
Password: ******
Password (again): ******
Superuser created successfully.

[...]
Django version 1.2.1, using settings 'celeryconfig'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Now that the service is started you can visit the monitor at http://127.0.0.1:8000, and log in using the user you created.

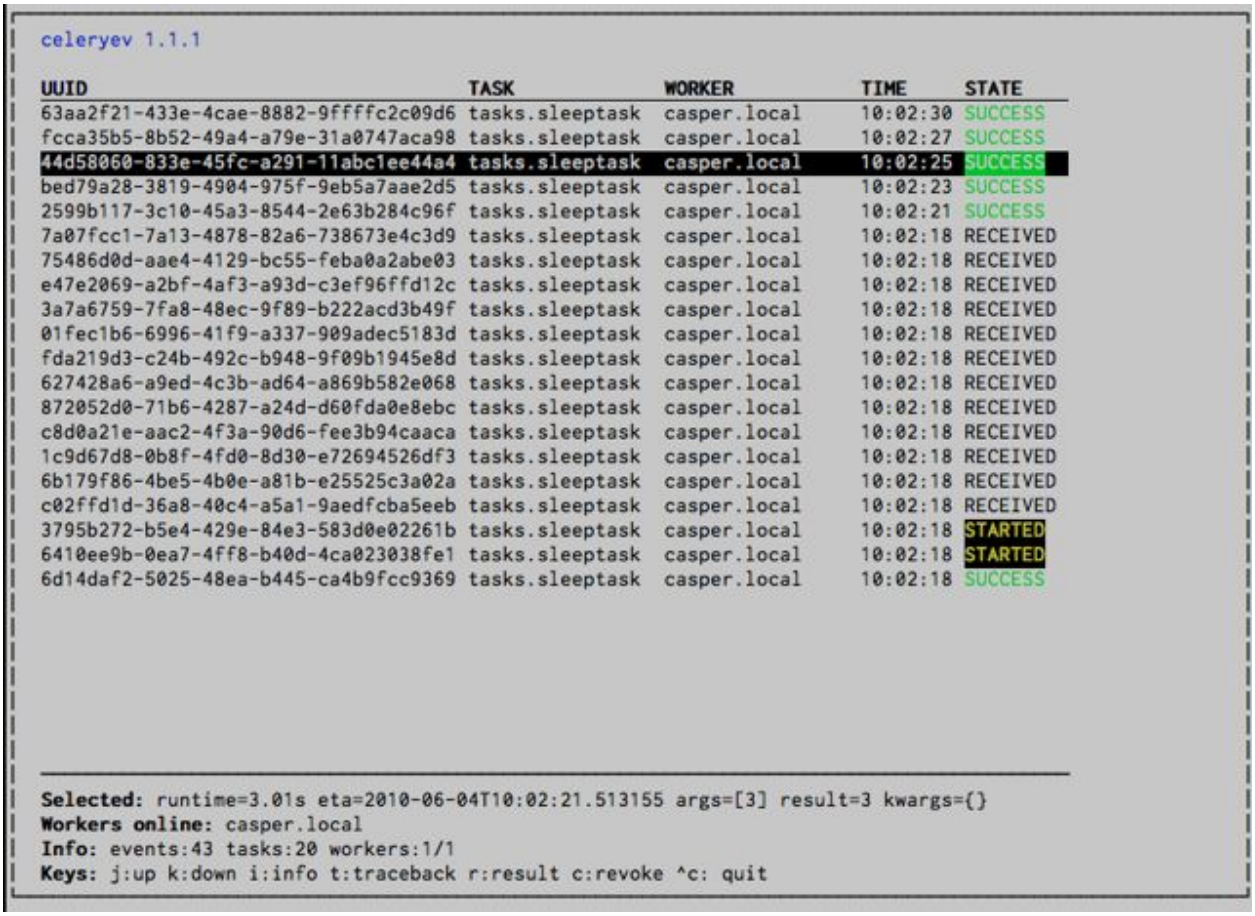
For a list of the command line options supported by **djcelerymon**, please see `djcelerymon --help`.

celeryev: Curses Monitor

New in version 2.0.

`celeryev` is a simple curses monitor displaying task and worker history. You can inspect the result and traceback of tasks, and it also supports some management commands like rate limiting and shutting down workers.

`celeryev` is also used to start snapshot cameras (see *Snapshots*:


celeryev 1.1.1
UUID TASK WORKER TIME STATE
63aa2f21-433e-4cae-8882-9ffffc2c09d6 tasks.sleeptask casper.local 10:02:30 SUCCESS
fcca35b5-8b52-49a4-a79e-31a0747aca98 tasks.sleeptask casper.local 10:02:27 SUCCESS
44d58060-833e-45fc-a291-11abc1ee44a4 tasks.sleeptask casper.local 10:02:25 SUCCESS
bed79a28-3819-4904-975f-9eb5a7aae2d5 tasks.sleeptask casper.local 10:02:23 SUCCESS
2599b117-3c10-45a3-8544-2e63b284c96f tasks.sleeptask casper.local 10:02:21 SUCCESS
7a07fcc1-7a13-4878-82a6-738673e4c3d9 tasks.sleeptask casper.local 10:02:18 RECEIVED
75486d0d-aae4-4129-bc55-feba0a2abe03 tasks.sleeptask casper.local 10:02:18 RECEIVED
e47e2069-a2bf-4af3-a93d-c3ef96ffd12c tasks.sleeptask casper.local 10:02:18 RECEIVED
3a7a6759-7fa8-48ec-9f89-b222acd3b49f tasks.sleeptask casper.local 10:02:18 RECEIVED
01fec1b6-6996-41f9-a337-909adec5183d tasks.sleeptask casper.local 10:02:18 RECEIVED
fda219d3-c24b-492c-b948-9f09b1945e8d tasks.sleeptask casper.local 10:02:18 RECEIVED
627428a6-a9ed-4c3b-ad64-a869b582e068 tasks.sleeptask casper.local 10:02:18 RECEIVED
872052d0-71b6-4287-a24d-d60fda0e8ebc tasks.sleeptask casper.local 10:02:18 RECEIVED
c8d0a21e-aac2-4f3a-90d6-fee3b94caaca tasks.sleeptask casper.local 10:02:18 RECEIVED
1c9d67d8-0b8f-4fd0-8d30-e72694526df3 tasks.sleeptask casper.local 10:02:18 RECEIVED
6b179f86-4be5-4b0e-a81b-e25525c3a02a tasks.sleeptask casper.local 10:02:18 RECEIVED
c02ffd1d-36a8-40c4-a5a1-9aedfcba5eeb tasks.sleeptask casper.local 10:02:18 RECEIVED
3795b272-b5e4-429e-84e3-583d0e02261b tasks.sleeptask casper.local 10:02:18 STARTED
6410ee9b-0ea7-4ff8-b40d-4ca023038fe1 tasks.sleeptask casper.local 10:02:18 STARTED
6d14daf2-5025-48ea-b445-ca4b9fcc9369 tasks.sleeptask casper.local 10:02:18 SUCCESS
Selected: runtime=3.01s eta=2010-06-04T10:02:21.513155 args=[3] result=3 kwargs={}
Workers online: casper.local
Info: events:43 tasks:20 workers:1/1
Keys: j:up k:down i:info t:traceback r:result c:revoke ^c: quit

```
$ celeryev --camera=<camera-class> --frequency=1.0
```

and it includes a tool to dump events to `stdout`:

```
$ celeryev --dump
```

For a complete list of options use `--help`:

```
$ celeryev --help
```

celerymon: Web monitor

celerymon is the ongoing work to create a web monitor. It's far from complete yet, and does currently only support a JSON API. Help is desperately needed for this project, so if you, or someone you know would like to contribute templates, design, code or help this project in any way, please get in touch!

> **Tip** The Django admin monitor can be used even though you're not using Celery with a Django project. See *Using outside of Django*.

2.9.3 RabbitMQ

To manage a Celery cluster it is important to know how RabbitMQ can be monitored.

RabbitMQ ships with the rabbitmqctl(1) command, with this you can list queues, exchanges, bindings, queue lengths, the memory usage of each queue, as well as manage users, virtual hosts and their permissions.

Note: The default virtual host (`"/"`) is used in these examples, if you use a custom virtual host you have to add the `-p` argument to the command, e.g: `rabbitmqctl list_queues -p my_vhost ....`

Inspecting queues

Finding the number of tasks in a queue:

```
$ rabbitmqctl list_queues name messages messages_ready \
                          messages_unacknowlged
```

Here *messages_ready* is the number of messages ready for delivery (sent but not received), *messages_unacknowledged* is the number of messages that has been received by a worker but not acknowledged yet (meaning it is in progress, or has been reserved). *messages* is the sum of ready and unacknowledged messages.

Finding the number of workers currently consuming from a queue:

```
$ rabbitmqctl list_queues name consumers
```

Finding the amount of memory allocated to a queue:

```
$ rabbitmqctl list_queues name memory
```

> **Tip** Adding the `-q` option to rabbitmqctl(1) makes the output easier to parse.

2.9.4 Munin

This is a list of known Munin plug-ins that can be useful when maintaining a Celery cluster.

- rabbitmq-munin: Munin plug-ins for RabbitMQ.

http://github.com/ask/rabbitmq-munin

- celery_tasks: Monitors the number of times each task type has been executed (requires *celerymon*).

  http://exchange.munin-monitoring.org/plugins/celery_tasks-2/details

- celery_task_states: Monitors the number of tasks in each state (requires *celerymon*).

  http://exchange.munin-monitoring.org/plugins/celery_tasks/details

2.9.5 Events

The worker has the ability to send a message whenever some event happens. These events are then captured by tools like **celerymon** and **celeryev** to monitor the cluster.

Snapshots

Even a single worker can produce a huge amount of events, so storing the history of all events on disk may be very expensive.

A sequence of events describes the cluster state in that time period, by taking periodic snapshots of this state we can keep all history, but still only periodically write it to disk.

To take snapshots you need a Camera class, with this you can define what should happen every time the state is captured; You can write it to a database, send it by email or something else entirely.

celeryev is then used to take snapshots with the camera, for example if you want to capture state every 2 seconds using the camera `myapp.Camera` you run **celeryev** with the following arguments:

```
$ celeryev -c myapp.Camera --frequency=2.0
```

Custom Camera

Here is an example camera, dumping the snapshot to screen:

```
from pprint import pformat

from celery.events.snapshot import Polaroid


class DumpCam(Polaroid):

    def on_shutter(self, state):
        if not state.event_count:
            # No new events since last snapshot.
            return
        print("Workers: %s" % (pformat(state.workers, indent=4), ))
        print("Tasks: %s" % (pformat(state.tasks, indent=4), ))
        print("Total: %s events, %s tasks" % (
            state.event_count, state.task_count))
```

See the API reference for `celery.events.state` to read more about state objects.

Now you can use this cam with **celeryev** by specifying it with the *-c* option:

```
$ celeryev -c myapp.DumpCam --frequency=2.0
```

Or you can use it programmatically like this:

```
from celery.events import EventReceiver
from celery.messaging import establish_connection
from celery.events.state import State
from myapp import DumpCam

def main():
    state = State()
    with establish_connection() as connection:
        recv = EventReceiver(connection, handlers={"*": state.event})
        with DumpCam(state, freq=1.0):
            recv.capture(limit=None, timeout=None)

if __name__ == "__main__":
    main()
```

Event Reference

This list contains the events sent by the worker, and their arguments.

Task Events

- `task-sent(uuid, name, args, kwargs, retries, eta, expires)`

  Sent when a task message is published and the `CELERY_SEND_TASK_SENT_EVENT` setting is enabled.

- `task-received(uuid, name, args, kwargs, retries, eta, hostname, timestamp)`

  Sent when the worker receives a task.

- `task-started(uuid, hostname, timestamp, pid)`

  Sent just before the worker executes the task.

- `task-succeeded(uuid, result, runtime, hostname, timestamp)`

  Sent if the task executed successfully.

  Runtime is the time it took to execute the task using the pool. (Starting from the task is sent to the worker pool, and ending when the pool result handler callback is called).

- `task-failed(uuid, exception, traceback, hostname, timestamp)`

  Sent if the execution of the task failed.

- `task-revoked(uuid)`

  Sent if the task has been revoked (Note that this is likely to be sent by more than one worker).

- `task-retried(uuid, exception, traceback, hostname, timestamp)`

  Sent if the task failed, but will be retried in the future.

Worker Events

- `worker-online(hostname, timestamp, sw_ident, sw_ver, sw_sys)`

  The worker has connected to the broker and is online.

  - *sw_ident*: Name of worker software (e.g. celeryd).

  - *sw_ver*: Software version (e.g. 2.2.0).
  - *sw_sys*: Operating System (e.g. Linux, Windows, Darwin).

- `worker-heartbeat(hostname, timestamp, sw_ident, sw_ver, sw_sys)`

  Sent every minute, if the worker has not sent a heartbeat in 2 minutes, it is considered to be offline.

- `worker-offline(hostname, timestamp, sw_ident, sw_ver, sw_sys)`

  The worker has disconnected from the broker.

2.10 Optimizing

2.10.1 Introduction

The default configuration makes a lot of compromises. It's not optimal for any single case, but works well enough for most situations.

There are optimizations that can be applied based on specific use cases.

Optimizations can apply to different properties of the running environment, be it the time tasks take to execute, the amount of memory used, or responsiveness at times of high load.

2.10.2 Ensuring Operations

In the book Programming Pearls, Jon Bentley presents the concept of back-of-the-envelope calculations by asking the question;

> How much water flows out of the Mississippi River in a day?

The point of this exercise[*] is to show that there is a limit to how much data a system can process in a timely manner. Back of the envelope calculations can be used as a means to plan for this ahead of time.

In Celery; If a task takes 10 minutes to complete, and there are 10 new tasks coming in every minute, the queue will never be empty. This is why it's very important that you monitor queue lengths!

A way to do this is by *using Munin*. You should set up alerts, that will notify you as soon as any queue has reached an unacceptable size. This way you can take appropriate action like adding new worker nodes, or revoking unnecessary tasks.

2.10.3 General Settings

Broker Connection Pools

You should enable the `BROKER_POOL_LIMIT` setting, as this will drastically improve overall performance.

The value should be based on how many threads will use connections at the same time.

This setting will be enabled by default in version 3.0.

2.10.4 Worker Settings

Prefetch Limits

Prefetch is a term inherited from AMQP that is often misunderstood by users.

The prefetch limit is a **limit** for the number of tasks (messages) a worker can reserve for itself. If it is zero, the worker will keep consuming messages, not respecting that there may be other available worker nodes that may be able to process them sooner[#], or that the messages may not even fit in memory.

The workers' default prefetch count is the `CELERYD_PREFETCH_MULTIPLIER` setting multiplied by the number of child worker processes[#].

If you have many tasks with a long duration you want the multiplier value to be 1, which means it will only reserve one task per worker process at a time.

However – If you have many short-running tasks, and throughput/round trip latency[#] is important to you, this number should be large. The worker is able to process more tasks per second if the messages have already been prefetched, and is available in memory. You may have to experiment to find the best value that works for you. Values like 50 or 150 might make sense in these circumstances. Say 64, or 128.

If you have a combination of long- and short-running tasks, the best option is to use two worker nodes that are configured separately, and route the tasks according to the run-time. (see *Routing Tasks*).

Reserve one task at a time

When using early acknowledgement (default), a prefetch multiplier of 1 means the worker will reserve at most one extra task for every active worker process.

When users ask if it's possible to disable "prefetching of tasks", often what they really want is to have a worker only reserve as many tasks as there are child processes.

But this is not possible without enabling late acknowledgements acknowledgements; A task that has been started, will be retried if the worker crashes mid execution so the task must be idempotent (see also notes at *Should I use retry or acks_late?*).

You can enable this behavior by using the following configuration options:

```
CELERY_ACKS_LATE = True
CELERYD_PREFETCH_MULTIPLIER = 1
```

Rate Limits

The system responsible for enforcing rate limits introduces some overhead, so if you're not using rate limits it may be a good idea to disable them completely. This will disable one thread, and it won't spend as many CPU cycles when the queue is inactive.

Set the `CELERY_DISABLE_RATE_LIMITS` setting to disable the rate limit subsystem:

```
CELERY_DISABLE_RATE_LIMITS = True
```

2.11 Concurrency

Release 2.4

Date February 04, 2014

2.11.1 Concurrency with Eventlet

Introduction

The Eventlet homepage describes it as; A concurrent networking library for Python that allows you to change how you run your code, not how you write it.

- It uses epoll(4) or libevent for highly scalable non-blocking I/O.
- Coroutines ensure that the developer uses a blocking style of programming that is similar to threading, but provide the benefits of non-blocking I/O.
- The event dispatch is implicit, which means you can easily use Eventlet from the Python interpreter, or as a small part of a larger application.

Celery supports Eventlet as an alternative execution pool implementation. It is in some cases superior to multiprocessing, but you need to ensure your tasks do not perform blocking calls, as this will halt all other operations in the worker until the blocking call returns.

The multiprocessing pool can take use of multiple processes, but how many is often limited to a few processes per CPU. With Eventlet you can efficiently spawn hundreds, or thousands of green threads. In an informal test with a feed hub system the Eventlet pool could fetch and process hundreds of feeds every second, while the multiprocessing pool spent 14 seconds processing 100 feeds. Note that is one of the applications evented I/O is especially good at (asynchronous HTTP requests). You may want a mix of both Eventlet and multiprocessing workers, and route tasks according to compatibility or what works best.

Enabling Eventlet

You can enable the Eventlet pool by using the `-P` option to **celeryd**:

```
$ celeryd -P eventlet -c 1000
```

Examples

See the Eventlet examples directory in the Celery distribution for some examples taking use of Eventlet support.

2.12 Signals

- Basics
- Signals
  - Task Signals
    - task_sent
    - task_prerun
    - task_postrun
    - task_failure
  - Worker Signals
    - worker_init
    - worker_ready
    - worker_process_init
    - worker_shutdown
  - Celerybeat Signals
    - beat_init
    - beat_embedded_init
  - Eventlet Signals
    - eventlet_pool_started
    - eventlet_pool_preshutdown
    - eventlet_pool_postshutdown
    - eventlet_pool_apply
  - Logging Signals
    - setup_logging
    - after_setup_logger
    - after_setup_task_logger

Signals allows decoupled applications to receive notifications when certain actions occur elsewhere in the application.

Celery ships with many signals that you application can hook into to augment behavior of certain actions.

2.12.1 Basics

Several kinds of events trigger signals, you can connect to these signals to perform actions as they trigger.

Example connecting to the `task_sent` signal:

```
from celery.signals import task_sent

def task_sent_handler(sender=None, task_id=None, task=None, args=None,
                      kwargs=None, \*\*kwds):
    print("Got signal task_sent for task id %s" % (task_id, ))

task_sent.connect(task_sent_handler)
```

Some signals also have a sender which you can filter by. For example the `task_sent` signal uses the task name as a sender, so you can connect your handler to be called only when tasks with name *"tasks.add"* has been sent by providing the *sender* argument to `connect`:

```
task_sent.connect(task_sent_handler, sender="tasks.add")
```

2.12.2 Signals

Task Signals

task_sent

Dispatched when a task has been sent to the broker. Note that this is executed in the client process, the one sending the task, not in the worker.

Sender is the name of the task being sent.

Provides arguments:

- **task_id** Id of the task to be executed.
- **task** The task being executed.
- **args** the tasks positional arguments.
- **kwargs** The tasks keyword arguments.
- **eta** The time to execute the task.
- **taskset** Id of the taskset this task is part of (if any).

task_prerun

Dispatched before a task is executed.

Sender is the task class being executed.

Provides arguments:

- **task_id** Id of the task to be executed.
- **task** The task being executed.
- **args** the tasks positional arguments.
- **kwargs** The tasks keyword arguments.

task_postrun

Dispatched after a task has been executed.

Sender is the task class executed.

Provides arguments:

- **task_id** Id of the task to be executed.
- **task** The task being executed.
- **args** The tasks positional arguments.
- **kwargs** The tasks keyword arguments.
- **retval** The return value of the task.

task_failure

Dispatched when a task fails.

Sender is the task class executed.

Provides arguments:

- **task_id** Id of the task.
- **exception** Exception instance raised.
- **args** Positional arguments the task was called with.
- **kwargs** Keyword arguments the task was called with.
- **traceback** Stack trace object.
- **einfo** The `celery.datastructures.ExceptionInfo` instance.

Worker Signals

worker_init

Dispatched before the worker is started.

worker_ready

Dispatched when the worker is ready to accept work.

worker_process_init

Dispatched by each new pool worker process when it starts.

worker_shutdown

Dispatched when the worker is about to shut down.

Celerybeat Signals

beat_init

Dispatched when celerybeat starts (either standalone or embedded). Sender is the `celery.beat.Service` instance.

beat_embedded_init

Dispatched in addition to the `beat_init` signal when celerybeat is started as an embedded process. Sender is the `celery.beat.Service` instance.

Eventlet Signals

eventlet_pool_started

Sent when the eventlet pool has been started.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

eventlet_pool_preshutdown

Sent when the worker shutdown, just before the eventlet pool is requested to wait for remaining workers.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

eventlet_pool_postshutdown

Sent when the pool has been joined and the worker is ready to shutdown.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

eventlet_pool_apply

Sent whenever a task is applied to the pool.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

Provides arguments:

- target

  The target function.

- args

  Positional arguments.

- kwargs

  Keyword arguments.

Logging Signals

setup_logging

Celery won't configure the loggers if this signal is connected, so you can use this to completely override the logging configuration with your own.

If you would like to augment the logging configuration setup by Celery then you can use the `after_setup_logger` and `after_setup_task_logger` signals.

Provides arguments:

- **loglevel** The level of the logging object.
- **logfile** The name of the logfile.
- **format** The log format string.

- **colorize** Specify if log messages are colored or not.

after_setup_logger

Sent after the setup of every global logger (not task loggers). Used to augment logging configuration.

Provides arguments:

- **logger** The logger object.
- **loglevel** The level of the logging object.
- **logfile** The name of the logfile.
- **format** The log format string.
- **colorize** Specify if log messages are colored or not.

after_setup_task_logger

Sent after the setup of every single task logger. Used to augment logging configuration.

Provides arguments:

- **logger** The logger object.
- **loglevel** The level of the logging object.
- **logfile** The name of the logfile.
- **format** The log format string.
- **colorize** Specify if log messages are colored or not.

CHAPTER 3

Configuration and defaults

This document describes the configuration options available.

If you're using the default loader, you must create the `celeryconfig.py` module and make sure it is available on the Python path.

- Example configuration file
- Configuration Directives
  - Concurrency settings
  - Task result backend settings
  - Database backend settings
  - AMQP backend settings
  - Cache backend settings
  - Tokyo Tyrant backend settings
  - Redis backend settings
  - MongoDB backend settings
  - Message Routing
  - Broker Settings
  - Task execution settings
  - Worker: celeryd
  - Error E-Mails
  - Events
  - Broadcast Commands
  - Logging
  - Custom Component Classes (advanced)
  - Periodic Task Server: celerybeat
  - Monitor Server: celerymon
  - Deprecated Settings

3.1 Example configuration file

This is an example configuration file to get you started. It should contain all you need to run a basic Celery set-up.

```
# List of modules to import when celery starts.
CELERY_IMPORTS = ("myapp.tasks", )

## Result store settings.
CELERY_RESULT_BACKEND = "database"
```

```
CELERY_RESULT_DBURI = "sqlite:///mydatabase.db"

## Broker settings.
BROKER_URL = "amqp://guest:guest@localhost:5672//"

## Worker settings
## If you're doing mostly I/O you can have more processes,
## but if mostly spending CPU, try to keep it close to the
## number of CPUs on your machine. If not set, the number of CPUs/cores
## available will be used.
CELERYD_CONCURRENCY = 10
```

3.2 Configuration Directives

3.2.1 Concurrency settings

CELERYD_CONCURRENCY

The number of concurrent worker processes/threads/green threads, executing tasks.

Defaults to the number of available CPUs.

CELERYD_PREFETCH_MULTIPLIER

How many messages to prefetch at a time multiplied by the number of concurrent processes. The default is 4 (four messages for each process). The default setting is usually a good choice, however – if you have very long running tasks waiting in the queue and you have to start the workers, note that the first worker to start will receive four times the number of messages initially. Thus the tasks may not be fairly distributed to the workers.

3.2.2 Task result backend settings

CELERY_RESULT_BACKEND

Deprecated aliases `CELERY_BACKEND`

The backend used to store task results (tombstones). Disabled by default. Can be one of the following:

- **database** Use a relational database supported by SQLAlchemy. See *Database backend settings*.
- **cache** Use memcached to store the results. See *Cache backend settings*.
- **mongodb** Use MongoDB to store the results. See *MongoDB backend settings*.
- **redis** Use Redis to store the results. See *Redis backend settings*.
- **tyrant** Use Tokyo Tyrant to store the results. See *Tokyo Tyrant backend settings*.
- **amqp** Send results back as AMQP messages See *AMQP backend settings*.

CELERY_RESULT_SERIALIZER

Result serialization format. Default is *"pickle"*. See *Serializers* for information about supported serialization formats.

3.2.3 Database backend settings

CELERY_RESULT_DBURI

Please see Supported Databases for a table of supported databases. To use this backend you need to configure it with an Connection String, some examples include:

```
# sqlite (filename)
CELERY_RESULT_DBURI = "sqlite:///celerydb.sqlite"

# mysql
CELERY_RESULT_DBURI = "mysql://scott:tiger@localhost/foo"

# postgresql
CELERY_RESULT_DBURI = "postgresql://scott:tiger@localhost/mydatabase"

# oracle
CELERY_RESULT_DBURI = "oracle://scott:tiger@127.0.0.1:1521/sidname"
```

See Connection String for more information about connection strings.

CELERY_RESULT_ENGINE_OPTIONS

To specify additional SQLAlchemy database engine options you can use the `CELERY_RESULT_ENGINE_OPTIONS` setting:

```
# echo enables verbose logging from SQLAlchemy.
CELERY_RESULT_ENGINE_OPTIONS = {"echo": True}
```

Short lived sessions are disabled by default. If enabled they can drastically reduce performance, especially on systems processing lots of tasks. This option is useful on low-traffic workers that experience errors as a result of cached database connections going stale through inactivity. For example, intermittent errors like *(OperationalError) (2006, 'MySQL server has gone away')* can be fixed by enabling short lived sessions. This option only affects the database backend.

Example configuration

```
CELERY_RESULT_BACKEND = "database"
CELERY_RESULT_DBURI = "mysql://user:password@host/dbname"
```

3.2.4 AMQP backend settings

Note: The AMQP backend requires RabbitMQ 1.1.0 or higher to automatically expire results. If you are running an older version of RabbitmQ you should disable result expiration like this:

CELERY_TASK_RESULT_EXPIRES = None

CELERY_RESULT_EXCHANGE

Name of the exchange to publish results in. Default is *"celeryresults"*.

CELERY_RESULT_EXCHANGE_TYPE

The exchange type of the result exchange. Default is to use a *direct* exchange.

CELERY_RESULT_PERSISTENT

If set to `True`, result messages will be persistent. This means the messages will not be lost after a broker restart. The default is for the results to be transient.

Example configuration

```
CELERY_RESULT_BACKEND = "amqp"
CELERY_TASK_RESULT_EXPIRES = 18000  # 5 hours.
```

3.2.5 Cache backend settings

Note: The cache backend supports the pylibmc and *python-memcached* libraries. The latter is used only if pylibmc is not installed.

CELERY_CACHE_BACKEND

Using a single memcached server:

```
CELERY_CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

Using multiple memcached servers:

```
CELERY_RESULT_BACKEND = "cache"
CELERY_CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11211/'
```

The "dummy" backend stores the cache in memory only:

CELERY_CACHE_BACKEND = "dummy"

CELERY_CACHE_BACKEND_OPTIONS

You can set pylibmc options using the `CELERY_CACHE_BACKEND_OPTIONS` setting:

```
CELERY_CACHE_BACKEND_OPTIONS = {"binary": True,
                                "behaviors": {"tcp_nodelay": True}}
```

3.2.6 Tokyo Tyrant backend settings

Note: The Tokyo Tyrant backend requires the `pytyrant` library: http://pypi.python.org/pypi/pytyrant/

This backend requires the following configuration directives to be set:

TT_HOST

Host name of the Tokyo Tyrant server.

TT_PORT

The port the Tokyo Tyrant server is listening to.

Example configuration

```
CELERY_RESULT_BACKEND = "tyrant"
TT_HOST = "localhost"
TT_PORT = 1978
```

3.2.7 Redis backend settings

Note: The Redis backend requires the `redis` library: http://pypi.python.org/pypi/redis/

To install the redis package use *pip* or *easy_install*:

```
$ pip install redis
```

This backend requires the following configuration directives to be set.

CELERY_REDIS_HOST

Host name of the Redis database server. e.g. *"localhost"*.

CELERY_REDIS_PORT

Port to the Redis database server. e.g. *6379*.

CELERY_REDIS_DB

Database number to use. Default is 0

CELERY_REDIS_PASSWORD

Password used to connect to the database.

Example configuration

```
CELERY_RESULT_BACKEND = "redis"
CELERY_REDIS_HOST = "localhost"
CELERY_REDIS_PORT = 6379
CELERY_REDIS_DB = 0
```

3.2.8 MongoDB backend settings

Note: The MongoDB backend requires the `pymongo` library: http://github.com/mongodb/mongo-python-driver/tree/master

CELERY_MONGODB_BACKEND_SETTINGS

This is a dict supporting the following keys:

- **host** Host name of the MongoDB server. Defaults to "localhost".
- **port** The port the MongoDB server is listening to. Defaults to 27017.
- **user** User name to authenticate to the MongoDB server as (optional).
- **password** Password to authenticate to the MongoDB server (optional).
- **database** The database name to connect to. Defaults to "celery".
- **taskmeta_collection** The collection name to store task meta data. Defaults to "celery_taskmeta".

Example configuration

```
CELERY_RESULT_BACKEND = "mongodb"
CELERY_MONGODB_BACKEND_SETTINGS = {
    "host": "192.168.1.100",
    "port": 30000,
    "database": "mydb",
    "taskmeta_collection": "my_taskmeta_collection",
}
```

3.2.9 Message Routing

CELERY_QUEUES

The mapping of queues the worker consumes from. This is a dictionary of queue name/options. See *Routing Tasks* for more information.

The default is a queue/exchange/binding key of *"celery"*, with exchange type *direct*.

You don't have to care about this unless you want custom routing facilities.

CELERY_ROUTES

A list of routers, or a single router used to route tasks to queues. When deciding the final destination of a task the routers are consulted in order. See *Routers* for more information.

CELERY_CREATE_MISSING_QUEUES

If enabled (default), any queues specified that is not defined in `CELERY_QUEUES` will be automatically created. See *Automatic routing*.

CELERY_DEFAULT_QUEUE

The queue used by default, if no custom queue is specified. This queue must be listed in `CELERY_QUEUES`. The default is: *celery*.

See also:

Changing the name of the default queue

CELERY_DEFAULT_EXCHANGE

Name of the default exchange to use when no custom exchange is specified. The default is: *celery*.

CELERY_DEFAULT_EXCHANGE_TYPE

Default exchange type used when no custom exchange is specified. The default is: *direct*.

CELERY_DEFAULT_ROUTING_KEY

The default routing key used when sending tasks. The default is: *celery*.

CELERY_DEFAULT_DELIVERY_MODE

Can be *transient* or *persistent*. The default is to send persistent messages.

3.2.10 Broker Settings

BROKER_TRANSPORT

Aliases `BROKER_BACKEND`

Deprecated aliases `CARROT_BACKEND`

The Kombu transport to use. Default is `amqplib`.

You can use a custom transport class name, or select one of the built-in transports: `amqplib`, `pika`, `redis`, `beanstalk`, `sqlalchemy`, `django`, `mongodb`, `couchdb`.

BROKER_URL

Default broker URL. This must be an URL in the form of:

```
transport://userid:password@hostname:port/virtual_host
```

Only the scheme part (`transport://`) is required, the rest is optional, and defaults to the specific transports default values.

If this setting is defined it will override a subset of the other `BROKER` options. These options are `BROKER_HOST`, `BROKER_USER`, `BROKER_PASSWORD`, `BROKER_PORT`, and `BROKER_VHOST`.

See the Kombu documentation for more information about broker URLs.

BROKER_HOST

Hostname of the broker.

BROKER_PORT

Custom port of the broker. Default is to use the default port for the selected backend.

BROKER_USER

Username to connect as.

BROKER_PASSWORD

Password to connect with.

BROKER_VHOST

Virtual host. Default is *"/"*.

BROKER_USE_SSL

Use SSL to connect to the broker. Off by default. This may not be supported by all transports.

BROKER_POOL_LIMIT

New in version 2.3.

The maximum number of connections that can be open in the connection pool.

A good default value could be 10, or more if you're using eventlet/gevent or lots of threads.

If set to `None` or 0 the connection pool will be disabled and connections will be established and closed for every use.

Disabled by default.

BROKER_CONNECTION_TIMEOUT

The default timeout in seconds before we give up establishing a connection to the AMQP server. Default is 4 seconds.

BROKER_CONNECTION_RETRY

Automatically try to re-establish the connection to the AMQP broker if lost.

The time between retries is increased for each retry, and is not exhausted before `BROKER_CONNECTION_MAX_RETRIES` is exceeded.

This behavior is on by default.

BROKER_CONNECTION_MAX_RETRIES

Maximum number of retries before we give up re-establishing a connection to the AMQP broker.

If this is set to `0` or `None`, we will retry forever.

Default is 100 retries.

BROKER_TRANSPORT_OPTIONS

New in version 2.2.

A dict of additional options passed to the underlying transport.

See your transport user manual for supported options (if any).

3.2.11 Task execution settings

CELERY_ALWAYS_EAGER

If this is `True`, all tasks will be executed locally by blocking until the task returns. `apply_async()` and `Task.delay()` will return an `EagerResult` instance, which emulates the API and behavior of `AsyncResult`, except the result is already evaluated.

That is, tasks will be executed locally instead of being sent to the queue.

CELERY_EAGER_PROPAGATES_EXCEPTIONS

If this is `True`, eagerly executed tasks (applied by *task.apply()*, or when the `CELERY_ALWAYS_EAGER` setting is enabled), will propagate exceptions.

It's the same as always running `apply()` with `throw=True`.

CELERY_IGNORE_RESULT

Whether to store the task return values or not (tombstones). If you still want to store errors, just not successful return values, you can set `CELERY_STORE_ERRORS_EVEN_IF_IGNORED`.

CELERY_MESSAGE_COMPRESSION

Default compression used for task messages. Can be `"gzip"`, `"bzip2"` (if available), or any custom compression schemes registered in the Kombu compression registry.

The default is to send uncompressed messages.

CELERY_TASK_RESULT_EXPIRES

Time (in seconds, or a `timedelta` object) for when after stored task tombstones will be deleted.

A built-in periodic task will delete the results after this time (`celery.task.backend_cleanup`).

Note: For the moment this only works with the amqp, database, cache, redis and MongoDB backends.

When using the database or MongoDB backends, *celerybeat* must be running for the results to be expired.

CELERY_MAX_CACHED_RESULTS

Result backends caches ready results used by the client.

This is the total number of results to cache before older results are evicted. The default is 5000.

CELERY_TRACK_STARTED

If `True` the task will report its status as "started" when the task is executed by a worker. The default value is `False` as the normal behaviour is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried. Having a "started" state can be useful for when there are long running tasks and there is a need to report which task is currently running.

CELERY_TASK_SERIALIZER

A string identifying the default serialization method to use. Can be *pickle* (default), *json*, *yaml*, *msgpack* or any custom serialization methods that have been registered with `kombu.serialization.registry`.

See also:

Serializers.

CELERY_TASK_PUBLISH_RETRY

New in version 2.2.

Decides if publishing task messages will be retried in the case of connection loss or other connection errors. See also `CELERY_TASK_PUBLISH_RETRY_POLICY`.

Disabled by default.

CELERY_TASK_PUBLISH_RETRY_POLICY

New in version 2.2.

Defines the default policy when retrying publishing a task message in the case of connection loss or other connection errors.

This is a mapping that must contain the following keys:

- *max_retries*

  Maximum number of retries before giving up, in this case the exception that caused the retry to fail will be raised.

  A value of 0 or `None` means it will retry forever.

  The default is to retry 3 times.

- *interval_start*

  Defines the number of seconds (float or integer) to wait between retries. Default is 0, which means the first retry will be instantaneous.

- *interval_step*

On each consecutive retry this number will be added to the retry delay (float or integer). Default is 0.2.

- *interval_max*

  Maximum number of seconds (float or integer) to wait between retries. Default is 0.2.

With the default policy of:

```
{"max_retries": 3,
 "interval_start": 0,
 "interval_step": 0.2,
 "interval_max": 0.2}
```

the maximum time spent retrying will be 0.4 seconds. It is set relatively short by default because a connection failure could lead to a retry pile effect if the broker connection is down: e.g. many web server processes waiting to retry blocking other incoming requests.

CELERY_DEFAULT_RATE_LIMIT

The global default rate limit for tasks.

This value is used for tasks that does not have a custom rate limit The default is no rate limit.

CELERY_DISABLE_RATE_LIMITS

Disable all rate limits, even if tasks has explicit rate limits set.

CELERY_ACKS_LATE

Late ack means the task messages will be acknowledged **after** the task has been executed, not *just before*, which is the default behavior.

See also:

FAQ: *Should I use retry or acks_late?*.

3.2.12 Worker: celeryd

CELERY_IMPORTS

A sequence of modules to import when the celery daemon starts.

This is used to specify the task modules to import, but also to import signal handlers and additional remote control commands, etc.

CELERYD_MAX_TASKS_PER_CHILD

Maximum number of tasks a pool worker process can execute before it's replaced with a new one. Default is no limit.

CELERYD_TASK_TIME_LIMIT

Task hard time limit in seconds. The worker processing the task will be killed and replaced with a new one when this is exceeded.

CELERYD_TASK_SOFT_TIME_LIMIT

Task soft time limit in seconds.

The `SoftTimeLimitExceeded` exception will be raised when this is exceeded. The task can catch this to e.g. clean up before the hard time limit comes.

Example:

```
from celery.task import task
from celery.exceptions import SoftTimeLimitExceeded

@task()
def mytask():
    try:
        return do_work()
    except SoftTimeLimitExceeded:
        cleanup_in_a_hurry()
```

CELERY_STORE_ERRORS_EVEN_IF_IGNORED

If set, the worker stores all task errors in the result store even if `Task.ignore_result` is on.

CELERYD_STATE_DB

Name of the file used to stores persistent worker state (like revoked tasks). Can be a relative or absolute path, but be aware that the suffix *.db* may be appended to the file name (depending on Python version).

Can also be set via the `--statedb` argument to `celeryd`.

Not enabled by default.

CELERYD_ETA_SCHEDULER_PRECISION

Set the maximum time in seconds that the ETA scheduler can sleep between rechecking the schedule. Default is 1 second.

Setting this value to 1 second means the schedulers precision will be 1 second. If you need near millisecond precision you can set this to 0.1.

3.2.13 Error E-Mails

CELERY_SEND_TASK_ERROR_EMAILS

The default value for the *Task.send_error_emails* attribute, which if set to `True` means errors occurring during task execution will be sent to `ADMINS` by email.

ADMINS

List of *(name, email_address)* tuples for the administrators that should receive error emails.

SERVER_EMAIL

The email address this worker sends emails from. Default is celery@localhost.

EMAIL_HOST

The mail server to use. Default is *"localhost"*.

EMAIL_HOST_USER

User name (if required) to log on to the mail server with.

EMAIL_HOST_PASSWORD

Password (if required) to log on to the mail server with.

EMAIL_PORT

The port the mail server is listening on. Default is *25*.

EMAIL_USE_SSL

Use SSL when connecting to the SMTP server. Disabled by default.

EMAIL_USE_TLS

Use TLS when connecting to the SMTP server. Disabled by default.

EMAIL_TIMEOUT

Timeout in seconds for when we give up trying to connect to the SMTP server when sending emails.

The default is 2 seconds.

Example E-Mail configuration

This configuration enables the sending of error emails to george@vandelay.com and kramer@vandelay.com:

```
# Enables error emails.
CELERY_SEND_TASK_ERROR_EMAILS = True

# Name and email addresses of recipients
ADMINS = (
    ("George Costanza", "george@vandelay.com"),
    ("Cosmo Kramer", "kosmo@vandelay.com"),
)

# Email address used as sender (From field).
SERVER_EMAIL = "no-reply@vandelay.com"
```

```
# Mailserver configuration
EMAIL_HOST = "mail.vandelay.com"
EMAIL_PORT = 25
# EMAIL_HOST_USER = "servers"
# EMAIL_HOST_PASSWORD = "s3cr3t"
```

3.2.14 Events

CELERY_SEND_EVENTS

Send events so the worker can be monitored by tools like *celerymon*.

CELERY_SEND_TASK_SENT_EVENT

New in version 2.2.

If enabled, a *task-sent* event will be sent for every task so tasks can be tracked before they are consumed by a worker.

Disabled by default.

CELERY_EVENT_SERIALIZER

Message serialization format used when sending event messages. Default is *"json"*. See *Serializers*.

3.2.15 Broadcast Commands

CELERY_BROADCAST_QUEUE

Name prefix for the queue used when listening for broadcast messages. The workers host name will be appended to the prefix to create the final queue name.

Default is *"celeryctl"*.

CELERY_BROADCAST_EXCHANGE

Name of the exchange used for broadcast messages.

Default is *"celeryctl"*.

CELERY_BROADCAST_EXCHANGE_TYPE

Exchange type used for broadcast messages. Default is *"fanout"*.

3.2.16 Logging

CELERYD_HIJACK_ROOT_LOGGER

New in version 2.2.

By default any previously configured logging options will be reset, because the Celery programs "hijacks" the root logger.

If you want to customize your own logging then you can disable this behavior.

Note: Logging can also be customized by connecting to the `celery.signals.setup_logging` signal.

CELERYD_LOG_COLOR

Enables/disables colors in logging output by the Celery apps.

By default colors are enabled if

1. the app is logging to a real terminal, and not a file.
2. the app is not running on Windows.

CELERYD_LOG_FORMAT

The format to use for log messages.

Default is *[%(asctime)s: %(levelname)s/%(processName)s] %(message)s*

See the Python `logging` module for more information about log formats.

CELERYD_TASK_LOG_FORMAT

The format to use for log messages logged in tasks. Can be overridden using the `--loglevel` option to `celeryd`.

Default is:

```
[%(asctime)s: %(levelname)s/%(processName)s]
    [%(task_name)s(%(task_id)s)] %(message)s
```

See the Python `logging` module for more information about log formats.

CELERY_REDIRECT_STDOUTS

If enabled *stdout* and *stderr* will be redirected to the current logger.

Enabled by default. Used by **celeryd** and **celerybeat**.

CELERY_REDIRECT_STDOUTS_LEVEL

The log level output to *stdout* and *stderr* is logged as. Can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL`.

Default is `WARNING`.

3.2.17 Custom Component Classes (advanced)

CELERYD_POOL

Name of the pool class used by the worker.

You can use a custom pool class name, or select one of the built-in aliases: `processes`, `eventlet`, `gevent`.

Default is `processes`.

CELERYD_AUTOSCALER

New in version 2.2.

Name of the autoscaler class to use.

Default is `"celery.worker.autoscale.Autoscaler"`.

CELERYD_CONSUMER

Name of the consumer class used by the worker. Default is `celery.worker.consumer.Consumer`

CELERYD_MEDIATOR

Name of the mediator class used by the worker. Default is `celery.worker.controllers.Mediator`.

CELERYD_ETA_SCHEDULER

Name of the ETA scheduler class used by the worker. Default is `celery.utils.timer2.Timer`, or one overrided by the pool implementation.

3.2.18 Periodic Task Server: celerybeat

CELERYBEAT_SCHEDULE

The periodic task schedule used by `celerybeat`. See *Entries*.

CELERYBEAT_SCHEDULER

The default scheduler class. Default is *"celery.beat.PersistentScheduler"*.

Can also be set via the `-S` argument to `celerybeat`.

CELERYBEAT_SCHEDULE_FILENAME

Name of the file used by *PersistentScheduler* to store the last run times of periodic tasks. Can be a relative or absolute path, but be aware that the suffix *.db* may be appended to the file name (depending on Python version).

Can also be set via the `--schedule` argument to `celerybeat`.

CELERYBEAT_MAX_LOOP_INTERVAL

The maximum number of seconds `celerybeat` can sleep between checking the schedule. Default is 300 seconds (5 minutes).

3.2.19 Monitor Server: celerymon

CELERYMON_LOG_FORMAT

The format to use for log messages.

Default is *[%(asctime)s: %(levelname)s/%(processName)s] %(message)s*

See the Python `logging` module for more information about log formats.

3.2.20 Deprecated Settings

These settings have been deprecated and should no longer used, as they will be removed in future versions.

CELERY_AMQP_TASK_RESULT_EXPIRES

Deprecated since version 2.5.

The time in seconds of which the task result queues should expire.

This setting is deprecated, and will be removed in version 3.0. Please use `CELERY_TASK_RESULT_EXPIRES` instead.

Note: AMQP result expiration requires RabbitMQ versions 2.1.0 and higher.

CELERY_TASK_ERROR_WHITELIST

Deprecated since version 2.5.

A white list of exceptions to send error emails for.

This option is pending deprecation and is scheduled for removal in version 3.0.

CELERYD_LOG_FILE

Deprecated since version 2.4.

This option is deprecated and is scheduled for removal in version 3.0. Please use the `--logfile` argument instead.

The default file name the worker daemon logs messages to. Can be overridden using the `--logfile` option to `celeryd`.

The default is `None` (*stderr*)

CELERYD_LOG_LEVEL

Deprecated since version 2.4.

This option is deprecated and is scheduled for removal in version 3.0. Please use the `--loglevel` argument instead.

Worker log level, can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL`.

Can also be set via the `--loglevel` argument to `celeryd`.

See the `logging` module for more information.

CELERYBEAT_LOG_FILE

Deprecated since version 2.4.

This option is deprecated and is scheduled for removal in version 3.0. Please use the `--logfile` argument instead.

The default file name to log messages to. Can be overridden using the *–logfile* option to `celerybeat`.

The default is `None` (*stderr*).

CELERYBEAT_LOG_LEVEL

Deprecated since version 2.4.

This option is deprecated and is scheduled for removal in version 3.0. Please use the `--loglevel` argument instead.

Logging level. Can be any of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`.

Can also be set via the `--loglevel` argument to `celerybeat`.

See the `logging` module for more information.

CELERYMON_LOG_FILE

Deprecated since version 2.4.

This option is deprecated and is scheduled for removal in version 3.0. Please use the `--logfile` argument instead.

The default file name to log messages to. Can be overridden using the `--logfile` argument to *celerymon*.

The default is `None` (*stderr*)

CELERYMON_LOG_LEVEL

Deprecated since version 2.4.

This option is deprecated and is scheduled for removal in version 3.0. Please use the `--loglevel` argument instead.

Logging level. Can be any of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`.

See the `logging` module for more information.

CHAPTER 4

Cookbook

4.1 Creating Tasks

- Ensuring a task is only executed one at a time

4.1.1 Ensuring a task is only executed one at a time

You can accomplish this by using a lock.

In this example we'll be using the cache framework to set a lock that is accessible for all workers.

It's part of an imaginary RSS feed importer called *djangofeeds*. The task takes a feed URL as a single argument, and imports that feed into a Django model called *Feed*. We ensure that it's not possible for two or more workers to import the same feed at the same time by setting a cache key consisting of the MD5 checksum of the feed URL.

The cache key expires after some time in case something unexpected happens (you never know, right?)

```
from celery.task import Task
from django.core.cache import cache
from django.utils.hashcompat import md5_constructor as md5
from djangofeeds.models import Feed

LOCK_EXPIRE = 60 * 5 # Lock expires in 5 minutes

class FeedImporter(Task):
    name = "feed.import"

    def run(self, feed_url, **kwargs):
        logger = self.get_logger(**kwargs)

        # The cache key consists of the task name and the MD5 digest
        # of the feed URL.
        feed_url_digest = md5(feed_url).hexdigest()
        lock_id = "%s-lock-%s" % (self.name, feed_url_hexdigest)

        # cache.add fails if if the key already exists
        acquire_lock = lambda: cache.add(lock_id, "true", LOCK_EXPIRE)
        # memcache delete is very slow, but we have to use it to take
        # advantage of using add() for atomic locking
        release_lock = lambda: cache.delete(lock_id)
```

```
        logger.debug("Importing feed: %s" % feed_url)
        if acquire_lock():
            try:
                feed = Feed.objects.import_feed(feed_url)
            finally:
                release_lock()
            return feed.url

        logger.debug(
            "Feed %s is already being imported by another worker" % (
                feed_url))
        return
```

4.2 Running celeryd as a daemon

Celery does not daemonize itself, please use one of the following daemonization tools.

- Generic init scripts
  - Init script: celeryd
    - Example configuration
    - Example Django configuration
    - Example Django configuration Using Virtualenv
    - Available options
  - Init script: celerybeat
    - Example configuration
    - Example Django configuration
    - Available options
  - Troubleshooting
- supervisord
- launchd (OS X)
- Windows

4.2.1 Generic init scripts

See the contrib/generic-init.d/ directory Celery distribution.

This directory contains generic bash init scripts for **celeryd**, that should run on Linux, FreeBSD, OpenBSD, and other Unix platforms.

Init script: celeryd

Usage */etc/init.d/celeryd {start|stop|restart|status}*

Configuration file /etc/default/celeryd

To configure celeryd you probably need to at least tell it where to change directory to when it starts (to find your *celeryconfig*).

Example configuration

This is an example configuration for a Python project.

`/etc/default/celeryd`:

```
# Name of nodes to start
# here we have a single node
CELERYD_NODES="w1"
# or we could have three nodes:
#CELERYD_NODES="w1 w2 w3"

# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# Extra arguments to celeryd
CELERYD_OPTS="--time-limit=300 --concurrency=8"

# Name of the celery config module.
CELERY_CONFIG_MODULE="celeryconfig"

# %n will be replaced with the nodename.
CELERYD_LOG_FILE="/var/log/celery/%n.log"
CELERYD_PID_FILE="/var/run/celery/%n.pid"

# Workers should run as an unprivileged user.
CELERYD_USER="celery"
CELERYD_GROUP="celery"
```

Example Django configuration

This is an example configuration for those using *django-celery*:

```
# Name of nodes to start, here we have a single node
CELERYD_NODES="w1"
# or we could have three nodes:
#CELERYD_NODES="w1 w2 w3"

# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# How to call "manage.py celeryd_multi"
CELERYD_MULTI="$CELERYD_CHDIR/manage.py celeryd_multi"

# How to call "manage.py celeryctl"
CELERYCTL="$CELERYD_CHDIR/manage.py celeryctl"

# Extra arguments to celeryd
CELERYD_OPTS="--time-limit=300 --concurrency=8"

# Name of the celery config module.
CELERY_CONFIG_MODULE="celeryconfig"

# %n will be replaced with the nodename.
CELERYD_LOG_FILE="/var/log/celery/%n.log"
CELERYD_PID_FILE="/var/run/celery/%n.pid"
```

```
# Workers should run as an unprivileged user.
CELERYD_USER="celery"
CELERYD_GROUP="celery"

# Name of the projects settings module.
export DJANGO_SETTINGS_MODULE="settings"
```

Example Django configuration Using Virtualenv

In case you are using virtualenv, you should add the path to your environment's python interpreter:

```
# Name of nodes to start, here we have a single node
CELERYD_NODES="w1"
# or we could have three nodes:
#CELERYD_NODES="w1 w2 w3"

# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# Python interpreter from environment.
ENV_PYTHON="$CELERYD_CHDIR/env/bin/python"

# How to call "manage.py celeryd_multi"
CELERYD_MULTI="$ENV_PYTHON $CELERYD_CHDIR/manage.py celeryd_multi"

# How to call "manage.py celeryctl"
CELERYCTL=$ENV_PYTHON $CELERYD_CHDIR/manage.py celeryctl"

# Extra arguments to celeryd
CELERYD_OPTS="--time-limit=300 --concurrency=8"

# Name of the celery config module.
CELERY_CONFIG_MODULE="celeryconfig"

# %n will be replaced with the nodename.
CELERYD_LOG_FILE="/var/log/celery/%n.log"
CELERYD_PID_FILE="/var/run/celery/%n.pid"

# Workers should run as an unprivileged user.
CELERYD_USER="celery"
CELERYD_GROUP="celery"

# Name of the projects settings module.
export DJANGO_SETTINGS_MODULE="settings"
```

Available options

- **CELERYD_NODES** Node names to start.
- **CELERYD_OPTS** Additional arguments to celeryd, see *celeryd –help* for a list.
- **CELERYD_CHDIR** Path to change directory to at start. Default is to stay in the current directory.
- **CELERYD_PID_FILE** Full path to the PID file. Default is /var/run/celeryd%n.pid
- **CELERYD_LOG_FILE** Full path to the celeryd log file. Default is /var/log/celeryd@%n.log

- **CELERYD_LOG_LEVEL** Log level to use for celeryd. Default is INFO.
- **CELERYD_MULTI** Path to the celeryd-multi program. Default is *celeryd-multi*. You can point this to a virtualenv, or even use manage.py for django.
- **CELERYCTL** Path to the celeryctl program. Default is *celeryctl*. You can point this to a virtualenv, or even use manage.py for django.
- **CELERYD_USER** User to run celeryd as. Default is current user.
- **CELERYD_GROUP** Group to run celeryd as. Default is current user.

Init script: celerybeat

Usage */etc/init.d/celerybeat {start|stop|restart}*

Configuration file /etc/default/celerybeat or /etc/default/celeryd

Example configuration

This is an example configuration for a Python project:

/etc/default/celerybeat:

```
# Where to chdir at start.
CELERYBEAT_CHDIR="/opt/Myproject/"

# Extra arguments to celerybeat
CELERYBEAT_OPTS="--schedule=/var/run/celerybeat-schedule"

# Name of the celery config module.#
CELERY_CONFIG_MODULE="celeryconfig"
```

Example Django configuration

This is an example configuration for those using *django-celery*

/etc/default/celerybeat:

```
# Where the Django project is.
CELERYBEAT_CHDIR="/opt/Project/"

# Name of the projects settings module.
export DJANGO_SETTINGS_MODULE="settings"

# Path to celerybeat
CELERYBEAT="/opt/Project/manage.py celerybeat"

# Extra arguments to celerybeat
CELERYBEAT_OPTS="--schedule=/var/run/celerybeat-schedule"
```

Available options

- **CELERYBEAT_OPTS** Additional arguments to celerybeat, see *celerybeat –help* for a list.
- **CELERYBEAT_PIDFILE** Full path to the PID file. Default is /var/run/celeryd.pid.

- **CELERYBEAT_LOGFILE** Full path to the celeryd log file. Default is /var/log/celeryd.log
- **CELERYBEAT_LOG_LEVEL** Log level to use for celeryd. Default is INFO.
- **CELERYBEAT** Path to the celeryd program. Default is *celeryd*. You can point this to an virtualenv, or even use manage.py for django.
- **CELERYBEAT_USER** User to run celeryd as. Default is current user.
- **CELERYBEAT_GROUP** Group to run celeryd as. Default is current user.

Troubleshooting

If you can't get the init scripts to work, you should try running them in *verbose mode*:

```
$ sh -x /etc/init.d/celeryd start
```

This can reveal hints as to why the service won't start.

Also you will see the commands generated, so you can try to run the celeryd command manually to read the resulting error output.

For example my *sh -x* output does this:

```
++ start-stop-daemon --start --chdir /opt/Opal/release/opal --quiet \
    --oknodo --background --make-pidfile --pidfile /var/run/celeryd.pid \
    --exec /opt/Opal/release/opal/manage.py celeryd -- --time-limit=300 \
    -f /var/log/celeryd.log -l INFO
```

Run the celeryd command after *–exec* (without the –) to show the actual resulting output:

```
$ /opt/Opal/release/opal/manage.py celeryd --time-limit=300 \
    -f /var/log/celeryd.log -l INFO
```

4.2.2 supervisord

- contrib/supervisord/

4.2.3 launchd (OS X)

- contrib/mac/

4.2.4 Windows

See this excellent external tutorial:

http://www.calazan.com/windows-tip-run-applications-in-the-background-using-task-scheduler/

This page contains common recipes and techniques.

CHAPTER 5

Contributing

- Community Code of Conduct
  - Be considerate.
  - Be respectful.
  - Be collaborative.
  - When you disagree, consult others.
  - When you are unsure, ask for help.
  - Step down considerately.
- Reporting a Bug
  - Issue Trackers
- Versions
- Branches
  - `master` branch
  - Maintenance branches
  - Archived branches
  - Feature branches
- Tags
- Working on Features & Patches
  - Forking and setting up the repository
  - Running the unit test suite
  - Creating pull requests
    - Calculating test coverage
    - Running the tests on all supported Python versions
  - Building the documentation
  - Verifying your contribution
    - pyflakes & PEP8
    - API reference
- Coding Style
- Release Procedure
  - Updating the version number
  - Releasing
  - Updating bundles

5.1 Community Code of Conduct

The goal is to maintain a diverse community that is pleasant for everyone. That is why we would greatly appreciate it if everyone contributing to and interacting with the community also followed this Code of Conduct.

The Code of Conduct covers our behavior as members of the community, in any forum, mailing list, wiki, website, Internet relay chat (IRC), public meeting or private correspondence.

The Code of Conduct is heavily based on the Ubuntu Code of Conduct, and the Pylons Code of Conduct.

5.1.1 Be considerate.

Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and we expect you to take those consequences into account when making decisions. Even if it's not obvious at the time, our contributions to Ubuntu will impact the work of others. For example, changes to code, infrastructure, policy, documentation and translations during a release may negatively impact others work.

5.1.2 Be respectful.

The Celery community and its members treat one another with respect. Everyone can make a valuable contribution to Celery. We may not always agree, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened is not a productive one. We expect members of the Celery community to be respectful when dealing with other contributors as well as with people outside the Celery project and with users of Celery.

5.1.3 Be collaborative.

Collaboration is central to Celery and to the larger free software community. We should always be open to collaboration. Your work should be done transparently and patches from Celery should be given back to the community when they are made, not just when the distribution releases. If you wish to work on new code for existing upstream projects, at least keep those projects informed of your ideas and progress. It many not be possible to get consensus from upstream, or even from your colleagues about the correct implementation for an idea, so don't feel obliged to have that agreement before you begin, but at least keep the outside world informed of your work, and publish your work in a way that allows outsiders to test, discuss and contribute to your efforts.

5.1.4 When you disagree, consult others.

Disagreements, both political and technical, happen all the time and the Celery community is no exception. It is important that we resolve disagreements and differing views constructively and with the help of the community and community process. If you really want to go a different way, then we encourage you to make a derivative distribution or alternate set of packages that still build on the work we've done to utilize as common of a core as possible.

5.1.5 When you are unsure, ask for help.

Nobody knows everything, and nobody is expected to be perfect. Asking questions avoids many problems down the road, and so questions are encouraged. Those who are asked questions should be responsive and helpful. However, when asking a question, care must be taken to do so in an appropriate forum.

5.1.6 Step down considerately.

Developers on every project come and go and Celery is no different. When you leave or disengage from the project, in whole or in part, we ask that you do so in a way that minimizes disruption to the project. This means you should tell people you are leaving and take the proper steps to ensure that others can pick up where you leave off.

5.2 Reporting a Bug

Bugs can always be described to the *Mailing list*, but the best way to report an issue and to ensure a timely response is to use the issue tracker.

1. Create a GitHub account.

You need to create a GitHub account to be able to create new issues and participate in the discussion.

2. Determine if your bug is really a bug.

You should not file a bug if you are requesting support. For that you can use the *Mailing list*, or *IRC*.

3. Make sure your bug hasn't already been reported.

Search through the appropriate Issue tracker. If a bug like yours was found, check if you have new information that could be reported to help the developers fix the bug.

4. Collect information about the bug.

To have the best chance of having a bug fixed, we need to be able to easily reproduce the conditions that caused it. Most of the time this information will be from a Python traceback message, though some bugs might be in design, spelling or other errors on the website/docs/code.

If the error is from a Python traceback, include it in the bug report.

We also need to know what platform you're running (Windows, OSX, Linux, etc), the version of your Python interpreter, and the version of Celery, and related packages that you were running when the bug occurred.

5. Submit the bug.

By default GitHub will email you to let you know when new comments have been made on your bug. In the event you've turned this feature off, you should check back on occasion to ensure you don't miss any questions a developer trying to fix the bug might ask.

5.2.1 Issue Trackers

Bugs for a package in the Celery ecosystem should be reported to the relevant issue tracker.

- Celery: http://github.com/ask/celery/issues/
- Django-Celery: http://github.com/ask/django-celery/issues
- Flask-Celery: http://github.com/ask/flask-celery/issues
- Celery-Pylons: http://bitbucket.org/ianschenck/celery-pylons/issues
- Kombu: http://github.com/ask/kombu/issues

If you are unsure of the origin of the bug you can ask the *Mailing list*, or just use the Celery issue tracker.

5.3 Versions

Version numbers consists of a major version, minor version and a release number. Since version 2.1.0 we use the versioning semantics described by semver: http://semver.org.

Stable releases are published at PyPI while development releases are only available in the GitHub git repository as tags. All version tags starts with "v", so version 0.8.0 is the tag v0.8.0.

5.4 Branches

Current active version branches:

- master (http://github.com/ask/celery/tree/master)
- 3.0-devel (http://github.com/ask/celery/tree/3.0-devel)

You can see the state of any branch by looking at the Changelog:

https://github.com/ask/celery/blob/master/Changelog

If the branch is in active development the topmost version info should contain metadata like:

```
2.4.0
======
:release-date: TBA
:status: DEVELOPMENT
:branch: master
```

The `status` field can be one of:

- `PLANNING`

  The branch is currently experimental and in the planning stage.

- `DEVELOPMENT`

  The branch is in active development, but the test suite should be passing and the product should be working and possible for users to test.

- `FROZEN`

  The branch is frozen, and no more features will be accepted. When a branch is frozen the focus is on testing the version as much as possible before it is released.

5.4.1 `master` branch

The master branch is where development of the next version happens.

5.4.2 Maintenance branches

Maintenance branches are named after the version, e.g. the maintenance branch for the 2.2.x series is named `2.2`. Previously these were named `releaseXX-maint`.

The versions we currently maintain is:

- 2.3

  This is the current series.

- 2.2

  This is the previous series, and the last version to support Python 2.4.

- 2.1

  This is the last version to use the `carrot` AMQP framework. Recent versions use `kombu`.

5.4.3 Archived branches

Archived branches are kept for preserving history only, and theoretically someone could provide patches for these if they depend on a series that is no longer officially supported.

An archived version is named `X.Y-archived`.

Our currently archived branches are:

- 2.1-archived
- 2.0-archived
- 1.0-archived

5.4.4 Feature branches

Major new features are worked on in dedicated branches. There is no strict naming requirement for these branches.

Feature branches are removed once they have been merged into a release branch.

5.5 Tags

Tags are used exclusively for tagging releases. A release tag is named with the format `vX.Y.Z`, e.g. `v2.3.1`. Experimental releases contain an additional identifier `vX.Y.Z-id`, e.g. `v3.0.0-rc1`. Experimental tags may be removed after the official release.

5.6 Working on Features & Patches

Note: Contributing to Celery should be as simple as possible, so none of these steps should be considered mandatory.

You can even send in patches by email if that is your preferred work method. We won't like you any less, any contribution you make is always appreciated!

However following these steps may make maintainers life easier, and may mean that your changes will be accepted sooner.

5.6.1 Forking and setting up the repository

First you need to fork the Celery repository, a good introduction to this is in the Github Guide: Fork a Repo.

After you have cloned the repository you should checkout your copy to a directory on your machine:

```
$ git clone git@github.com:username/celery.git
```

When the repository is cloned enter the directory to set up easy access to upstream changes:

```
$ cd celery
$ git remote add upstream git://github.com/ask/celery.git
$ git fetch upstream
```

If you need to pull in new changes from upstream you should always use the `--rebase` option to `git pull`:

```
git pull --rebase upstream master
```

With this option you don't clutter the history with merging commit notes. See Rebasing merge commits in git. If you want to learn more about rebasing see the Rebase section in the Github guides.

If you need to work on a different branch than `master` you can fetch and checkout a remote branch like this:

```
git checkout --track -b 3.0-devel origin/3.0-devel
```

For a list of branches see *Branches*.

5.6.2 Running the unit test suite

To run the Celery test suite you need to install a few dependencies. A complete list of the dependencies needed are located in `requirements/test.txt`.

Installing the test requirements:

```
$ pip -E $VIRTUAL_ENV install -U -r requirements/test.txt
```

When installation of dependencies is complete you can execute the test suite by calling `nosetests`:

```
$ nosetests
```

Some useful options to **nosetests** are:

- *-x*

  Stop running the tests at the first test that fails.

- *-s*

  Don't capture output

- *--nologcapture*

  Don't capture log output.

- *-v*

  Run with verbose output.

If you want to run the tests for a single test file only you can do so like this:

```
$ nosetests celery.tests.test_worker.test_worker_job
```

5.6.3 Creating pull requests

When your feature/bugfix is complete you may want to submit a pull requests so that it can be reviewed by the maintainers.

Creating pull requests is easy, and also let you track the progress of your contribution. Read the Pull Requests section in the Github Guide to learn how this is done.

You can also attach pull requests to existing issues by following the steps outlined here: http://bit.ly/koJoso

Calculating test coverage

Code coverage in HTML:

```
$ nosetests --with-coverage3 --cover3-html
```

The coverage output will then be located at `celery/tests/cover/index.html`.

Code coverage in XML (Cobertura-style):

```
$ nosetests --with-coverage3 --cover3-xml --cover3-xml-file=coverage.xml
```

The coverage XML output will then be located at `coverage.xml`

Running the tests on all supported Python versions

There is a `tox` configuration file in the top directory of the distribution.

To run the tests for all supported Python versions simply execute:

```
$ tox
```

If you only want to test specific Python versions use the *-e* option:

```
$ tox -e py25,py26
```

5.6.4 Building the documentation

To build the documentation you need to install the dependencies listed in `requirements/docs.txt`:

```
$ pip -E $VIRTUAL_ENV install -U -r requirements/docs.txt
```

After these dependencies are installed you should be able to build the docs by running:

```
$ cd docs
$ rm -rf .build
$ make html
```

Make sure there are no errors or warnings in the build output. After building succeeds the documentation is available at `.build/html`.

5.6.5 Verifying your contribution

To use these tools you need to install a few dependencies. These dependencies can be found in `requirements/pkgutils.txt`.

Installing the dependencies:

```
$ pip -E $VIRTUAL_ENV install -U -r requirements/pkgutils.txt
```

pyflakes & PEP8

To ensure that your changes conform to PEP8 and to run pyflakes execute:

```
$ paver flake8
```

To not return a negative exit code when this command fails use the *-E* option, this can be convenient while developing:

```
$ paver flake8 -E
```

API reference

To make sure that all modules have a corresponding section in the API reference please execute:

```
$ paver autodoc
$ paver verifyindex
```

If files are missing you can add them by copying an existing reference file.

If the module is internal it should be part of the internal reference located in `docs/internals/reference/`. If the module is public it should be located in `docs/reference/`.

For example if reference is missing for the module `celery.worker.awesome` and this module is considered part of the public API, use the following steps:

```
$ cd docs/reference/
$ cp celery.schedules.rst celery.worker.awesome.rst
$ vim celery.worker.awesome.rst

    # change every occurance of ``celery.schedules`` to
    # ``celery.worker.awesome``

$ vim index.rst

    # Add ``celery.worker.awesome`` to the index.

# Add the file to git
$ git add celery.worker.awesome.rst
$ git add index.rst
$ git commit celery.worker.awesome.rst index.rst \
    -m "Adds reference for celery.worker.awesome"
```

5.7 Coding Style

You should probably be able to pick up the coding style from surrounding code, but it is a good idea to be aware of the following conventions.

- All Python code must follow the PEP-8 guidelines.

pep8.py is an utility you can use to verify that your code is following the conventions.

- Docstrings must follow the PEP-257 conventions, and use the following style.

  Do this:

  ```
  def method(self, arg):
      """Short description.

      More details.

      """
  ```

  or:

```
def method(self, arg):
    """Short description."""
```

but not this:

```
def method(self, arg):
    """
    Short description.
    """
```

- Lines should not exceed 78 columns.

  You can enforce this in **vim** by setting the `textwidth` option:

  ```
  set textwidth=78
  ```

  If adhering to this limit makes the code less readable, you have one more character to go on, which means 78 is a soft limit, and 79 is the hard limit :)

- Import order
  - Python standard library (*import xxx*)
  - Python standard library ('from xxx import')
  - Third party packages.
  - Other modules from the current package.

  or in case of code using Django:
  - Python standard library (*import xxx*)
  - Python standard library ('from xxx import')
  - Third party packages.
  - Django packages.
  - Other modules from the current package.

  Within these sections the imports should be sorted by module name.

  Example:

  ```
  import threading
  import time

  from collections import deque
  from Queue import Queue, Empty

  from .datastructures import TokenBucket
  from .utils import timeutils
  from .utils.compat import all, izip_longest, chain_from_iterable
  ```

- Wildcard imports must not be used (*from xxx import **).
- For distributions where Python 2.5 is the oldest support version additional rules apply:
  - Absolute imports must be enabled at the top of every module:

    ```
    from __future__ import absolute_import
    ```

  - If the module uses the with statement it must also enable that:

```
from __future__ import with_statement
```

– Every future import must be on its own line, as older Python 2.5 releases did not support importing multiple features on the same future import line:

```
# Good
from __future__ import absolute_import
from __future__ import with_statement

# Bad
from __future__ import absolute_import, with_statement
```

(Note that this rule does not apply if the package does not include support for Python 2.5)

- Note that we use "new-style' relative imports when the distribution does not support Python versions below 2.5

```
from . import submodule
```

5.8 Release Procedure

5.8.1 Updating the version number

The version number must be updated two places:

- `celery/__init__.py`
- `docs/include/introduction.txt`

After you have changed these files you must render the `README` files. There is a script to convert sphinx syntax to generic reStructured Text syntax, and the paver task *readme* does this for you:

```
$ paver readme
```

Now commit the changes:

```
$ git commit -a -m "Bumps version to X.Y.Z"
```

and make a new version tag:

```
$ git tag vX.Y.Z
$ git push --tags
```

5.8.2 Releasing

Commands to make a new public stable release:

```
$ paver releaseok     # checks pep8, autodoc index and runs tests
$ paver removepyc  # Remove .pyc files.
$ git clean -xdn # Check that there's no left-over files in the repository.
$ python2.5 setup.py sdist upload # Upload package to PyPI
$ paver upload_pypi_docs
$ paver ghdocs # Build and upload documentation to Github.
```

If this is a new release series then you also need to do the following:

- **Go to the Read The Docs management interface at:** http://readthedocs.org/projects/celery/?fromdocs=celery

- Enter "Edit project"

  Change default branch to the branch of this series, e.g. `2.4` for series 2.4.

- Also add the previous version under the "versions" tab.

5.8.3 Updating bundles

First you need to make sure the bundle entrypoints have been installed, but either running *develop*, or *install*:

```
$ python setup.py develop
```

Then make sure that you have your PyPI credentials stored in `~/.pypirc`, and execute the command:

```
$ python setup.py upload_bundles
```

If you broke something and need to update new versions of the bundles, then you can use `upload_bundles_fix`.

CHAPTER 6

Community Resources

This is a list of external blog posts, tutorials and slides related to Celery. If you have a link that's missing from this list, please contact the mailing-list or submit a patch.

- Resources
  - Who's using Celery
  - Wiki
  - Celery questions on Stack Overflow
  - Mailing-list Archive: celery-users
  - IRC Logs
- News
  - Celery: Mini Tutorial (Catalan)
  - Building a Django App Server with Chef
  - Introducció a Celery (Catalan)
  - Django and Celery - Death to Cron
  - Celery Tips
  - What's your favorite Django app?
  - Virtualenv Tips
  - 10 Tools That Make Django Better
  - Distributed Task Locking in Celery
  - Celery — Python (Russian)
  - Celery (Russian)
  - Celery, RabbitMQ and sending messages directly.
  - Cron dentro do Django com Celery (Portugese)
  - RabbitMQCeleryDjango (Japanese)
  - Kaninchen & Schlangen: RabbitMQ & Python (German)
  - Celery - Eine asynchrone Task Queue (nicht nur) für Django (German)
  - Asynchronous Processing Using Celery (historio.us)
  - "Massaging the Pony: Message Queues and You" (Djangocon 2010)
  - "Large problems, Mostly Solved" (Djangocon 2010)
  - A Simple Celery with Django How-To
  - Django and asynchronous jobs
  - Celery: Django (Russian)
  - Celery presentation at PyCon India 2010
  - celery, django and virtualenv playing nice.
  - Django Task Queueing with Celery
  - django-celery presentation at DJUGL 2010.
  - Django/Celery Quickstart (or, how I learned to stop using cron and love celery)
  - Using Python magic to improve the deferred API
  - How Celery, Carrot, and your messaging stack work
  - Large Problems in Django, Mostly Solved: Delayed Execution
  - Introduction to Celery
  - RabbitMQ, Celery and Django
  - Message Queues, Django and Celery Quick Start
  - Background task processing and deferred execution in Django
  - Build a processing queue [...] in less than a day using RabbitMQ and Celery
  - How to get celeryd to work on FreeBSD
  - Web-based 3D animation software
  - Queued Storage Backend for Django
  - RabbitMQ with Python and Ruby

6.1 Resources

6.1.1 Who's using Celery

http://wiki.github.com/ask/celery/using

6.1.2 Wiki

http://wiki.github.com/ask/celery/

6.1.3 Celery questions on Stack Overflow

http://stackoverflow.com/search?q=celery&tab=newest

6.1.4 Mailing-list Archive: celery-users

http://blog.gmane.org/gmane.comp.python.amqp.celery.user

6.1.5 IRC Logs

http://botland.oebfare.com/logger/celery/

6.2 News

6.2.1 Celery: Mini Tutorial (Catalan)

http://trespams.com/2010/11/28/celery-mini-tutorial/

6.2.2 Building a Django App Server with Chef

http://ericholscher.com/blog/2010/nov/11/building-django-app-server-chef-part-4/

6.2.3 Introducció a Celery (Catalan)

http://trespams.com/2010/11/13/introduccio-celery/

6.2.4 Django and Celery - Death to Cron

http://tensixtyone.com/django-and-celery-death-to-cron

6.2.5 Celery Tips

http://ericholscher.com/blog/2010/nov/2/celery-tips/

6.2.6 What's your favorite Django app?

http://jacobian.org/writing/favorite-apps/

6.2.7 Virtualenv Tips

http://ericholscher.com/blog/2010/nov/1/virtualenv-tips/

6.2.8 10 Tools That Make Django Better

http://iamseb.com/seb/2010/10/10-django-tools/

6.2.9 Distributed Task Locking in Celery

http://www.loose-bits.com/2010_10_10_archive.html

6.2.10 Celery — Python (Russian)

http://www.bitbybit.ru/article/216

6.2.11 Celery (Russian)

http://vorushin.ru/blog/34-celery-described/

6.2.12 Celery, RabbitMQ and sending messages directly.

http://blog.timc3.com/2010/10/17/celery-rabbitmq-and-sending-messages-directly/

6.2.13 Cron dentro do Django com Celery (Portugese)

http://blog.avelino.us/2010/10/cron-dentro-do-django-com-celery.html

6.2.14 RabbitMQCeleryDjango (Japanese)

http://d.hatena.ne.jp/yuku_t/

6.2.15 Kaninchen & Schlangen: RabbitMQ & Python (German)

http://www.scribd.com/doc/37562923/Kaninchen-Schlangen-RabbitMQ-Python

6.2.16 Celery - Eine asynchrone Task Queue (nicht nur) für Django (German)

http://www.scribd.com/doc/39203296/Celery-Eine-asynchrone-Task-Queue-nicht-nur-fur-Django

6.2.17 Asynchronous Processing Using Celery (historio.us)

http://blog.historio.us/asynchronous-processing-using-celery

6.2.18 "Massaging the Pony: Message Queues and You" (Djangocon 2010)

http://www.slideshare.net/shawnrider/massaging-the-pony-message-queues-and-you

6.2.19 "Large problems, Mostly Solved" (Djangocon 2010)

http://www.slideshare.net/ericholscher/large-problems

6.2.20 A Simple Celery with Django How-To

http://shawnmilo.blogspot.com/2010/07/simple-celery-with-django-how-to.html

6.2.21 Django and asynchronous jobs

http://www.davidfischer.name/2010/09/django-and-asynchronous-jobs/

6.2.22 Celery: Django (Russian)

http://www.proft.com.ua/2010/09/4/celery-dobavlyaem-parallelizm-v-django/

6.2.23 Celery presentation at PyCon India 2010

http://in.pycon.org/2010/talks/50-python-celery http://in.pycon.org/2010/static/files/talks/50/mahendra-celery-pycon-2010.pdf

6.2.24 celery, django and virtualenv playing nice.

http://tumblr.whatupderek.com/post/1072002968/celery-django-and-virtualenv-playing-nice

6.2.25 Django Task Queueing with Celery

http://justinvoss.wordpress.com/2010/06/22/django-task-queueing-with-celery/

6.2.26 django-celery presentation at DJUGL 2010.

http://www.slideshare.net/matclayton/django-celery

6.2.27 Django/Celery Quickstart (or, how I learned to stop using cron and love celery)

http://bitkickers.blogspot.com/2010/07/djangocelery-quickstart-or-how-i.html

6.2.28 Using Python magic to improve the deferred API

http://blog.notdot.net/2010/06/Using-Python-magic-to-improve-the-deferred-API

6.2.29 How Celery, Carrot, and your messaging stack work

http://jasonmbaker.com/how-celery-carrot-and-your-messaging-stack-wo

6.2.30 Large Problems in Django, Mostly Solved: Delayed Execution

http://ericholscher.com/blog/2010/jun/23/large-problems-django-mostly-solved-delayed-execut/

6.2.31 Introduction to Celery

Awesome slides from when Idan Gazit had a talk about Celery at PyWeb-IL: http://www.slideshare.net/idangazit/an-introduction-to-celery

6.2.32 RabbitMQ, Celery and Django

Great Celery tutorial by Robert Pogorzelski at his blog "Happy Stream of Thoughts": http://robertpogorzelski.com/blog/2009/09/10/rabbitmq-celery-and-django/

6.2.33 Message Queues, Django and Celery Quick Start

Celery tutorial by Rich Leland, the installation section is Mac OS X specific: http://mathematism.com/2010/feb/16/message-queues-django-and-celery-quick-start/

6.2.34 Background task processing and deferred execution in Django

Alon Swartz writes about celery and RabbitMQ on his blog: http://www.turnkeylinux.org/blog/django-celery-rabbitmq

6.2.35 Build a processing queue [...] in less than a day using RabbitMQ and Celery

Tutorial in 2 parts written by Tim Bull: http://timbull.com/build-a-processing-queue-with-multi-threading

6.2.36 How to get celeryd to work on FreeBSD

Installing multiprocessing on FreeBSD isn't that easy, but thanks to Viktor Petersson we now have a step-to-step guide: http://www.playingwithwire.com/2009/10/how-to-get-celeryd-to-work-on-freebsd/

6.2.37 Web-based 3D animation software

Indy Chang Liu at ThinkingCactus uses Celery to render animations asynchronously (PDF): http://ojs.pythonpapers.org/index.php/tppm/article/viewFile/105/122

6.2.38 Queued Storage Backend for Django

http://stepsandnumbers.com/archive/2010/01/04/queued-storage-backend-for-django/

6.2.39 RabbitMQ with Python and Ruby

http://www.slideshare.net/hungryblank/rabbitmq-with-python-and-ruby-rupy-2009

CHAPTER 7

Tutorials

Release 2.4

Date February 04, 2014

7.1 Using Celery with Redis/Database as the messaging queue.

7.1.1 Redis

This section has been moved to *Using Redis*.

7.1.2 SQLAlchemy

This section has been moved to *Using SQLAlchemy*.

7.1.3 Django Database

This section has been moved to *Using the Django Database*.

7.2 Debugging Tasks Remotely (using pdb)

7.2.1 Basics

`celery.contrib.rdb` is an extended version of `pdb` that enables remote debugging of processes that does not have terminal access.

Example usage:

```
from celery.contrib import rdb
from celery.task import task

@task
def add(x, y):
    result = x + y
    rdb.set_trace()  # <- set breakpoint
    return result
```

`set_trace()` sets a breakpoint at the current location and creates a socket you can telnet into to remotely debug your task.

The debugger may be started by multiple processes at the same time, so rather than using a fixed port the debugger will search for an available port, starting from the base port (6900 by default). The base port can be changed using the environment variable `CELERY_RDB_PORT`.

By default the debugger will only be available from the local host, to enable access from the outside you have to set the environment variable `CELERY_RDB_HOST`.

When *celeryd* encounters your breakpoint it will log the following information:

```
[INFO/MainProcess] Got task from broker:
    tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8]
[WARNING/PoolWorker-1] Remote Debugger:6900:
    Please telnet 127.0.0.1 6900.  Type `exit` in session to continue.
[2011-01-18 14:25:44,119: WARNING/PoolWorker-1] Remote Debugger:6900:
    Waiting for client...
```

If you telnet the port specified you will be presented with a *pdb* shell:

```
$ telnet localhost 6900
Connected to localhost.
Escape character is '^]'.
> /opt/devel/demoapp/tasks.py(128)add()
-> return result
(Pdb)
```

Enter `help` to get a list of available commands, It may be a good idea to read the Python Debugger Manual if you have never used *pdb* before.

To demonstrate, we will read the value of the `result` variable, change it and continue execution of the task:

```
(Pdb) result
4
(Pdb) result = "hello from rdb"
(Pdb) continue
Connection closed by foreign host.
```

The result of our vandalism can be seen in the worker logs:

```
[2011-01-18 14:35:36,599: INFO/MainProcess] Task
    tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8] succeeded
    in 61.481s: 'hello from rdb'
```

7.2.2 Tips

Enabling the breakpoint signal

If the environment variable `CELERY_RDBSIG` is set, the worker will open up an rdb instance whenever the *SIGUSR2* signal is sent. This is the case for both main and worker processes.

For example starting the worker with:

```
CELERY_RDBSIG=1 celeryd -l info
```

You can start an rdb session for any of the worker processes by executing:

```
kill -USR2 <pid>
```

7.3 Tutorial: Creating a click counter using Kombu and celery

- Introduction
- The model
- Using Kombu to send clicks as messages
- View and URLs
- Creating the periodic task
- Finishing

7.3.1 Introduction

A click counter should be easy, right? Just a simple view that increments a click in the DB and forwards you to the real destination.

This would work well for most sites, but when traffic starts to increase, you are likely to bump into problems. One database write for every click is not good if you have millions of clicks a day.

So what can you do? In this tutorial we will send the individual clicks as messages using *kombu*, and then process them later with a Celery periodic task.

Celery and Kombu are excellent in tandem, and while this might not be the perfect example, you'll at least see one example how of they can be used to solve a task.

7.3.2 The model

The model is simple, *Click* has the URL as primary key and a number of clicks for that URL. Its manager, *ClickManager* implements the *increment_clicks* method, which takes a URL and by how much to increment its count by.

clickmuncher/models.py:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class ClickManager(models.Manager):

    def increment_clicks(self, for_url, increment_by=1):
        """Increment the click count for an URL.

            >>> Click.objects.increment_clicks("http://google.com", 10)

        """
        click, created = self.get_or_create(url=for_url,
                                    defaults={"click_count": increment_by})
        if not created:
            click.click_count += increment_by
            click.save()

        return click.click_count
```

```
class Click(models.Model):
    url = models.URLField(_(u"URL"), verify_exists=False, unique=True)
    click_count = models.PositiveIntegerField(_(u"click_count"),
                                              default=0)

    objects = ClickManager()

    class Meta:
        verbose_name = _(u"URL clicks")
        verbose_name_plural = _(u"URL clicks")
```

7.3.3 Using Kombu to send clicks as messages

The model is normal django stuff, nothing new there. But now we get on to the messaging. It has been a tradition for me to put the projects messaging related code in its own *messaging.py* module, and I will continue to do so here so maybe you can adopt this practice. In this module we have two functions:

- *send_increment_clicks*

  This function sends a simple message to the broker. The message body only contains the URL we want to increment as plain-text, so the exchange and routing key play a role here. We use an exchange called *clicks*, with a routing key of *increment_click*, so any consumer binding a queue to this exchange using this routing key will receive these messages.

- *process_clicks*

  This function processes all currently gathered clicks sent using *send_increment_clicks*. Instead of issuing one database query for every click it processes all of the messages first, calculates the new click count and issues one update per URL. A message that has been received will not be deleted from the broker until it has been acknowledged by the receiver, so if the receiver dies in the middle of processing the message, it will be re-sent at a later point in time. This guarantees delivery and we respect this feature here by not acknowledging the message until the clicks has actually been written to disk.

  Note: This could probably be optimized further with some hand-written SQL, but it will do for now. Let's say it's an exercise left for the picky reader, albeit a discouraged one if you can survive without doing it.

On to the code...

clickmuncher/messaging.py:

```
from celery.messaging import establish_connection
from kombu.compat import Publisher, Consumer
from clickmuncher.models import Click


def send_increment_clicks(for_url):
    """Send a message for incrementing the click count for an URL."""
    connection = establish_connection()
    publisher = Publisher(connection=connection,
                          exchange="clicks",
                          routing_key="increment_click",
                          exchange_type="direct")

    publisher.send(for_url)

    publisher.close()
    connection.close()
```

```
def process_clicks():
    """Process all currently gathered clicks by saving them to the
    database."""
    connection = establish_connection()
    consumer = Consumer(connection=connection,
                        queue="clicks",
                        exchange="clicks",
                        routing_key="increment_click",
                        exchange_type="direct")

    # First process the messages: save the number of clicks
    # for every URL.
    clicks_for_url = {}
    messages_for_url = {}
    for message in consumer.iterqueue():
        url = message.body
        clicks_for_url[url] = clicks_for_url.get(url, 0) + 1
        # We also need to keep the message objects so we can ack the
        # messages as processed when we are finished with them.
        if url in messages_for_url:
            messages_for_url[url].append(message)
        else:
            messages_for_url[url] = [message]

    # Then increment the clicks in the database so we only need
    # one UPDATE/INSERT for each URL.
    for url, click_count in clicks_for_urls.items():
        Click.objects.increment_clicks(url, click_count)
        # Now that the clicks has been registered for this URL we can
        # acknowledge the messages
        [message.ack() for message in messages_for_url[url]]

    consumer.close()
    connection.close()
```

7.3.4 View and URLs

This is also simple stuff, don't think I have to explain this code to you. The interface is as follows, if you have a link to http://google.com you would want to count the clicks for, you replace the URL with:

http://mysite/clickmuncher/count/?u=http://google.com

and the *count* view will send off an increment message and forward you to that site.

clickmuncher/views.py:

```
from django.http import HttpResponseRedirect
from clickmuncher.messaging import send_increment_clicks


def count(request):
    url = request.GET["u"]
    send_increment_clicks(url)
    return HttpResponseRedirect(url)
```

clickmuncher/urls.py:

```
from django.conf.urls.defaults import patterns, url
from clickmuncher import views

urlpatterns = patterns("",
    url(r'^$', views.count, name="clickmuncher-count"),
)
```

7.3.5 Creating the periodic task

Processing the clicks every 30 minutes is easy using celery periodic tasks.

clickmuncher/tasks.py:

```
from celery.task import PeriodicTask
from clickmuncher.messaging import process_clicks
from datetime import timedelta


class ProcessClicksTask(PeriodicTask):
    run_every = timedelta(minutes=30)

    def run(self, **kwargs):
        process_clicks()
```

We subclass from `celery.task.base.PeriodicTask`, set the *run_every* attribute and in the body of the task just call the *process_clicks* function we wrote earlier.

7.3.6 Finishing

There are still ways to improve this application. The URLs could be cleaned so the URL http://google.com and http://google.com/ is the same. Maybe it's even possible to update the click count using a single UPDATE query?

If you have any questions regarding this tutorial, please send a mail to the mailing-list or come join us in the #celery IRC channel at Freenode: http://celeryq.org/introduction.html#getting-help

CHAPTER 8

Frequently Asked Questions

- General
  - What kinds of things should I use Celery for?
- Misconceptions
  - Is Celery dependent on pickle?
  - Is Celery for Django only?
  - Do I have to use AMQP/RabbitMQ?
  - Is Celery multilingual?
- Troubleshooting
  - MySQL is throwing deadlock errors, what can I do?
  - celeryd is not doing anything, just hanging
  - Task results aren't reliably returning
  - Why is Task.delay/apply*/celeryd just hanging?
  - Does it work on FreeBSD?
  - I'm having *IntegrityError: Duplicate Key* errors. Why?
  - Why aren't my tasks processed?
  - Why won't my Task run?
  - Why won't my periodic task run?
  - How do I discard all waiting tasks?
  - I've discarded messages, but there are still messages left in the queue?
- Results
  - How do I get the result of a task if I have the ID that points there?
- Security
  - Isn't using *pickle* a security concern?
  - Can messages be encrypted?
  - Is it safe to run **celeryd** as root?
- Brokers
  - Why is RabbitMQ crashing?
  - Can I use Celery with ActiveMQ/STOMP?
  - What features are not supported when not using an AMQP broker?
- Tasks
  - How can I reuse the same connection when applying tasks?
  - Can I execute a task by name?
  - How can I get the task id of the current task?
  - Can I specify a custom task_id?
  - Can I use decorators with tasks?
  - Can I use natural task ids?
  - How can I run a task once another task has finished?
  - Can I cancel the execution of a task?
  - Why aren't my remote control commands received by all workers?
  - Can I send some tasks to only some servers?
  - Can I change the interval of a periodic task at runtime?
  - Does celery support task priorities?
  - Should I use retry or acks_late?
  - Can I schedule tasks to execute at a specific time?
  - How do I shut down *celeryd* safely?
  - How do I run celeryd in the background on [platform]?
- Windows
  - celeryd keeps spawning processes at startup
  - The *-B* / *–beat* option to celeryd doesn't work?
  - *django-celery* can't find settings?

8.1 General

8.1.1 What kinds of things should I use Celery for?

Answer: Queue everything and delight everyone is a good article describing why you would use a queue in a web context.

These are some common use cases:

- Running something in the background. For example, to finish the web request as soon as possible, then update the users page incrementally. This gives the user the impression of good performance and "snappiness", even though the real work might actually take some time.
- Running something after the web request has finished.
- Making sure something is done, by executing it asynchronously and using retries.
- Scheduling periodic work.

And to some degree:

- Distributed computing.
- Parallel execution.

8.2 Misconceptions

8.2.1 Is Celery dependent on pickle?

Answer: No.

Celery can support any serialization scheme and has built-in support for JSON, YAML, Pickle and msgpack. Also, as every task is associated with a content type, you can even send one task using pickle, and another using JSON.

The default serialization format is pickle simply because it is convenient (it supports sending complex Python objects as task arguments).

If you need to communicate with other languages you should change to a serialization format that is suitable for that.

You can set a global default serializer, the default serializer for a particular Task, or even what serializer to use when sending a single task instance.

8.2.2 Is Celery for Django only?

Answer: No.

Celery does not depend on Django anymore. To use Celery with Django you have to use the django-celery package.

8.2.3 Do I have to use AMQP/RabbitMQ?

Answer: No.

You can also use Redis, Beanstalk, CouchDB, MongoDB or an SQL database, see Using other queues.

These "virtual transports" may have limited broadcast and event functionality. For example remote control commands only works with AMQP and Redis.

Redis or a database won't perform as well as an AMQP broker. If you have strict reliability requirements you are encouraged to use RabbitMQ or another AMQP broker. Redis/database also use polling, so they are likely to consume more resources. However, if you for some reason are not able to use AMQP, feel free to use these alternatives. They will probably work fine for most use cases, and note that the above points are not specific to Celery; If using Redis/database as a queue worked fine for you before, it probably will now. You can always upgrade later if you need to.

8.2.4 Is Celery multilingual?

Answer: Yes.

`celeryd` is an implementation of Celery in Python. If the language has an AMQP client, there shouldn't be much work to create a worker in your language. A Celery worker is just a program connecting to the broker to process messages.

Also, there's another way to be language independent, and that is to use REST tasks, instead of your tasks being functions, they're URLs. With this information you can even create simple web servers that enable preloading of code. See: User Guide: Remote Tasks.

8.3 Troubleshooting

8.3.1 MySQL is throwing deadlock errors, what can I do?

Answer: MySQL has default isolation level set to *REPEATABLE-READ*, if you don't really need that, set it to *READ-COMMITTED*. You can do that by adding the following to your `my.cnf`:

```
[mysqld]
transaction-isolation = READ-COMMITTED
```

For more information about InnoDB's transaction model see MySQL - The InnoDB Transaction Model and Locking in the MySQL user manual.

(Thanks to Honza Kral and Anton Tsigularov for this solution)

8.3.2 celeryd is not doing anything, just hanging

Answer: See MySQL is throwing deadlock errors, what can I do?. or *Why is Task.delay/apply* just hanging?*.

8.3.3 Task results aren't reliably returning

Answer: If you're using the database backend for results, and in particular using MySQL, see MySQL is throwing deadlock errors, what can I do?.

8.3.4 Why is Task.delay/apply*/celeryd just hanging?

Answer: There is a bug in some AMQP clients that will make it hang if it's not able to authenticate the current user, the password doesn't match or the user does not have access to the virtual host specified. Be sure to check your broker logs (for RabbitMQ that is `/var/log/rabbitmq/rabbit.log` on most systems), it usually contains a message describing the reason.

8.3.5 Does it work on FreeBSD?

Answer: The multiprocessing pool requires a working POSIX semaphore implementation which isn't enabled in FreeBSD by default. You have to enable POSIX semaphores in the kernel and manually recompile multiprocessing.

Luckily, Viktor Petersson has written a tutorial to get you started with Celery on FreeBSD here: http://www.playingwithwire.com/2009/10/how-to-get-celeryd-to-work-on-freebsd/

8.3.6 I'm having *IntegrityError: Duplicate Key* errors. Why?

Answer: See MySQL is throwing deadlock errors, what can I do?. Thanks to howsthedotcom.

8.3.7 Why aren't my tasks processed?

Answer: With RabbitMQ you can see how many consumers are currently receiving tasks by running the following command:

```
$ rabbitmqctl list_queues -p <myvhost> name messages consumers
Listing queues ...
celery     2891    2
```

This shows that there's 2891 messages waiting to be processed in the task queue, and there are two consumers processing them.

One reason that the queue is never emptied could be that you have a stale worker process taking the messages hostage. This could happen if celeryd wasn't properly shut down.

When a message is received by a worker the broker waits for it to be acknowledged before marking the message as processed. The broker will not re-send that message to another consumer until the consumer is shut down properly.

If you hit this problem you have to kill all workers manually and restart them:

```
ps auxww | grep celeryd | awk '{print $2}' | xargs kill
```

You might have to wait a while until all workers have finished the work they're doing. If it's still hanging after a long time you can kill them by force with:

```
ps auxww | grep celeryd | awk '{print $2}' | xargs kill -9
```

8.3.8 Why won't my Task run?

Answer: There might be syntax errors preventing the tasks module being imported.

You can find out if Celery is able to run the task by executing the task manually:

```
>>> from myapp.tasks import MyPeriodicTask
>>> MyPeriodicTask.delay()
```

Watch celeryd's log file to see if it's able to find the task, or if some other error is happening.

8.3.9 Why won't my periodic task run?

Answer: See Why won't my Task run?.

8.3.10 How do I discard all waiting tasks?

Answer: You can use celeryctl to purge all configured task queues:

```
$ celeryctl purge
```

or programatically:

```
>>> from celery.task.control import discard_all
>>> discard_all()
1753
```

If you only want to purge messages from a specific queue you have to use the AMQP API or the **camqadm** utility:

```
$ camqadm queue.purge <queue name>
```

The number 1753 is the number of messages deleted.

You can also start `celeryd` with the `--purge` argument, to purge messages when the worker starts.

8.3.11 I've discarded messages, but there are still messages left in the queue?

Answer: Tasks are acknowledged (removed from the queue) as soon as they are actually executed. After the worker has received a task, it will take some time until it is actually executed, especially if there are a lot of tasks already waiting for execution. Messages that are not acknowledged are held on to by the worker until it closes the connection to the broker (AMQP server). When that connection is closed (e.g. because the worker was stopped) the tasks will be re-sent by the broker to the next available worker (or the same worker when it has been restarted), so to properly purge the queue of waiting tasks you have to stop all the workers, and then discard the tasks using `discard_all()`.

8.4 Results

8.4.1 How do I get the result of a task if I have the ID that points there?

Answer: Use *Task.AsyncResult*:

```
>>> result = MyTask.AsyncResult(task_id)
>>> result.get()
```

This will give you a `BaseAsyncResult` instance using the tasks current result backend.

If you need to specify a custom result backend you should use `celery.result.BaseAsyncResult` directly:

```
>>> from celery.result import BaseAsyncResult
>>> result = BaseAsyncResult(task_id, backend=...)
>>> result.get()
```

8.5 Security

8.5.1 Isn't using *pickle* a security concern?

Answer: Yes, indeed it is.

You are right to have a security concern, as this can indeed be a real issue. It is essential that you protect against unauthorized access to your broker, databases and other services transmitting pickled data.

For the task messages you can set the `CELERY_TASK_SERIALIZER` setting to "json" or "yaml" instead of pickle. There is currently no alternative solution for task results (but writing a custom result backend using JSON is a simple task)

Note that this is not just something you should be aware of with Celery, for example also Django uses pickle for its cache client.

8.5.2 Can messages be encrypted?

Answer: Some AMQP brokers supports using SSL (including RabbitMQ). You can enable this using the `BROKER_USE_SSL` setting.

It is also possible to add additional encryption and security to messages, if you have a need for this then you should contact the *Mailing list*.

8.5.3 Is it safe to run celeryd as root?

Answer: No!

We're not currently aware of any security issues, but it would be incredibly naive to assume that they don't exist, so running the Celery services (**celeryd**, **celerybeat**, **celeryev**, etc) as an unprivileged user is recommended.

8.6 Brokers

8.6.1 Why is RabbitMQ crashing?

Answer: RabbitMQ will crash if it runs out of memory. This will be fixed in a future release of RabbitMQ. please refer to the RabbitMQ FAQ: http://www.rabbitmq.com/faq.html#node-runs-out-of-memory

---

Note: This is no longer the case, RabbitMQ versions 2.0 and above includes a new persister, that is tolerant to out of memory errors. RabbitMQ 2.1 or higher is recommended for Celery.

If you're still running an older version of RabbitMQ and experience crashes, then please upgrade!

Misconfiguration of Celery can eventually lead to a crash on older version of RabbitMQ. Even if it doesn't crash, this can still consume a lot of resources, so it is very important that you are aware of the common pitfalls.

- Events.

Running `celeryd` with the `-E`/`--events` option will send messages for events happening inside of the worker.

Events should only be enabled if you have an active monitor consuming them, or if you purge the event queue periodically.

- AMQP backend results.

When running with the AMQP result backend, every task result will be sent as a message. If you don't collect these results, they will build up and RabbitMQ will eventually run out of memory.

Results expire after 1 day by default. It may be a good idea to lower this value by configuring the `CELERY_TASK_RESULT_EXPIRES` setting.

If you don't use the results for a task, make sure you set the *ignore_result* option:

8.6.2 Can I use Celery with ActiveMQ/STOMP?

Answer: No. It used to be supported by Carrot, but is not currently supported in Kombu.

8.6.3 What features are not supported when not using an AMQP broker?

This is an incomplete list of features not available when using the virtual transports:

- Remote control commands (supported only by Redis).
- Monitoring with events may not work in all virtual transports.
- **The *header* and *fanout* exchange types** (*fanout* is supported by Redis).

8.7 Tasks

8.7.1 How can I reuse the same connection when applying tasks?

Answer: See the `BROKER_POOL_LIMIT` setting. This setting will be enabled by default in 3.0.

8.7.2 Can I execute a task by name?

Answer: Yes. Use `celery.execute.send_task()`. You can also execute a task by name from any language that has an AMQP client.

```
>>> from celery.execute import send_task
>>> send_task("tasks.add", args=[2, 2], kwargs={})
<AsyncResult: 373550e8-b9a0-4666-bc61-ace01fa4f91d>
```

8.7.3 How can I get the task id of the current task?

Answer: The current id and more is available in the task request:

```
@task
def mytask():
    cache.set(mytask.request.id, "Running")
```

For more information see *Context*.

8.7.4 Can I specify a custom task_id?

Answer: Yes. Use the *task_id* argument to `apply_async()`:

```
>>> task.apply_async(args, kwargs, task_id="...")
```

8.7.5 Can I use decorators with tasks?

Answer: Yes. But please see note at *Decorating tasks*.

8.7.6 Can I use natural task ids?

Answer: Yes, but make sure it is unique, as the behavior for two tasks existing with the same id is undefined.

The world will probably not explode, but at the worst they can overwrite each others results.

8.7.7 How can I run a task once another task has finished?

Answer: You can safely launch a task inside a task. Also, a common pattern is to use callback tasks:

```
@task()
def add(x, y, callback=None):
    result = x + y
    if callback:
        subtask(callback).delay(result)
    return result


@task(ignore_result=True)
def log_result(result, **kwargs):
    logger = log_result.get_logger(**kwargs)
    logger.info("log_result got: %s" % (result, ))
```

Invocation:

```
>>> add.delay(2, 2, callback=log_result.subtask())
```

See *Sets of tasks, Subtasks and Callbacks* for more information.

8.7.8 Can I cancel the execution of a task?

Answer: Yes. Use *result.revoke*:

```
>>> result = add.apply_async(args=[2, 2], countdown=120)
>>> result.revoke()
```

or if you only have the task id:

```
>>> from celery.task.control import revoke
>>> revoke(task_id)
```

8.7.9 Why aren't my remote control commands received by all workers?

Answer: To receive broadcast remote control commands, every worker node uses its host name to create a unique queue name to listen to, so if you have more than one worker with the same host name, the control commands will be received in round-robin between them.

To work around this you can explicitly set the host name for every worker using the `--hostname` argument to `celeryd`:

```
$ celeryd --hostname=$(hostname).1
$ celeryd --hostname=$(hostname).2
```

etc., etc...

8.7.10 Can I send some tasks to only some servers?

Answer: Yes. You can route tasks to an arbitrary server using AMQP, and a worker can bind to as many queues as it wants.

See *Routing Tasks* for more information.

8.7.11 Can I change the interval of a periodic task at runtime?

Answer: Yes. You can use the Django database scheduler, or you can override *PeriodicTask.is_due* or turn *PeriodicTask.run_every* into a property:

```
class MyPeriodic(PeriodicTask):

    def run(self):
        # ...

    @property
    def run_every(self):
        return get_interval_from_database(...)
```

8.7.12 Does celery support task priorities?

Answer: No. In theory, yes, as AMQP supports priorities. However RabbitMQ doesn't implement them yet.

The usual way to prioritize work in Celery, is to route high priority tasks to different servers. In the real world this may actually work better than per message priorities. You can use this in combination with rate limiting to achieve a highly responsive system.

8.7.13 Should I use retry or acks_late?

Answer: Depends. It's not necessarily one or the other, you may want to use both.

Task.retry is used to retry tasks, notably for expected errors that is catchable with the *try:* block. The AMQP transaction is not used for these errors: **if the task raises an exception it is still acknowledged!**.

The *acks_late* setting would be used when you need the task to be executed again if the worker (for some reason) crashes mid-execution. It's important to note that the worker is not known to crash, and if it does it is usually an unrecoverable error that requires human intervention (bug in the worker, or task code).

In an ideal world you could safely retry any task that has failed, but this is rarely the case. Imagine the following task:

```
@task()
def process_upload(filename, tmpfile):
    # Increment a file count stored in a database
    increment_file_counter()
    add_file_metadata_to_db(filename, tmpfile)
    copy_file_to_destination(filename, tmpfile)
```

If this crashed in the middle of copying the file to its destination the world would contain incomplete state. This is not a critical scenario of course, but you can probably imagine something far more sinister. So for ease of programming we have less reliability; It's a good default, users who require it and know what they are doing can still enable acks_late (and in the future hopefully use manual acknowledgement)

In addition *Task.retry* has features not available in AMQP transactions: delay between retries, max retries, etc.

So use retry for Python errors, and if your task is idempotent combine that with *acks_late* if that level of reliability is required.

8.7.14 Can I schedule tasks to execute at a specific time?

Answer: Yes. You can use the *eta* argument of `Task.apply_async()`.

Or to schedule a periodic task at a specific time, use the `celery.schedules.crontab` schedule behavior:

```
from celery.task.schedules import crontab
from celery.task import periodic_task

@periodic_task(run_every=crontab(hours=7, minute=30, day_of_week="mon"))
def every_monday_morning():
    print("This is run every Monday morning at 7:30")
```

8.7.15 How do I shut down *celeryd* safely?

Answer: Use the `TERM` signal, and the worker will finish all currently executing jobs and shut down as soon as possible. No tasks should be lost.

You should never stop `celeryd` with the `KILL` signal (`-9`), unless you've tried `TERM` a few times and waited a few minutes to let it get a chance to shut down. As if you do tasks may be terminated mid-execution, and they will not be re-run unless you have the *acks_late* option set (*Task.acks_late* / `CELERY_ACKS_LATE`).

See also:

Stopping the worker

8.7.16 How do I run celeryd in the background on [platform]?

Answer: Please see *Running celeryd as a daemon*.

8.8 Windows

8.8.1 celeryd keeps spawning processes at startup

Answer: This is a known issue on Windows. You have to start celeryd with the command:

```
$ python -m celeryd.bin.celeryd
```

Any additional arguments can be appended to this command.

See http://bit.ly/bo9RSw

8.8.2 The *-B* / *–beat* option to celeryd doesn't work?

Answer: That's right. Run *celerybeat* and *celeryd* as separate services instead.

8.8.3 *django-celery* can't find settings?

Answer: You need to specify the `--settings` argument to **manage.py**:

```
$ python manage.py celeryd start --settings=settings
```

See http://bit.ly/bo9RSw

CHAPTER 9

Change history

- 2.4.7
- 2.4.6
- 2.4.5
- 2.4.4
  - Security Fixes
  - Fixes
- 2.4.3
- 2.4.2
- 2.4.1
- 2.4.0
  - Important Notes
  - News
- 2.3.4
  - Security Fixes
  - Fixes
- 2.3.3
- 2.3.2
  - News
  - Fixes
- 2.3.1
  - Fixes
- 2.3.0
  - Important Notes
  - News
  - Fixes
- 2.2.8
  - Security Fixes
- 2.2.7
- 2.2.6
  - Important Notes
  - Fixes
- 2.2.5
  - Important Notes
  - News
  - Fixes
- 2.2.4
  - Fixes
- 2.2.3
  - Fixes
- 2.2.2
  - Fixes
- 2.2.1
  - Fixes
- 2.2.0
  - Important Notes
  - News
  - Fixes
  - Experimental
- 2.1.4
  - Fixes
  - Documentation
- 2.1.3
- 2.1.2
  - Fixes
- 2.1.1
  - Fixes
  - News
- 2.1.0
  - Important Notes
  - News

9.1 2.4.7

release-date 2012-07-10 11:30 P.M GMT

- This release only changes the Kombu dependency so that version 2.2 is not installed (Issue #850).

9.2 2.4.6

release-date 2011-12-28 12:20 P.M GMT

- The group id was not changed if both `--uid` and `--gid` specified.

9.3 2.4.5

release-date 2011-12-02 05:00 P.M GMT

by Ask Solem

- Periodic task interval schedules were accidentally rounded down, resulting in some periodic tasks being executed early.
- Logging of humanized times in the celerybeat log is now more detailed.
- New *Brokers* section in the Getting Started part of the Documentation

  This replaces the old *Using Celery with Redis/Database as the messaging queue.* tutorial, and adds documentation for MongoDB, Beanstalk and CouchDB.

9.4 2.4.4

release-date 2011-11-25 16:00 P.M GMT

by Ask Solem

9.4.1 Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/`--gid` arguments to **celeryd-multi**, **celeryd_detach**, **celerybeat** and **celeryev** were used.

  This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

9.4.2 Fixes

- Processes pool: Fixed rare deadlock at shutdown (Issue #523).

  Fix contributed by Ionel Maries Christian.

- Webhook tasks issued the wrong HTTP POST headers (Issue #515).

The *Content-Type* header has been changed from `application/json` `application/x-www-form-urlencoded`, and adds a proper *Content-Length* header.

Fix contributed by Mitar.

- Daemonization cookbook: Adds a configuration example using Django and virtualenv together (Issue #505).

  Contributed by Juan Ignacio Catalano.

- generic init scripts now automatically creates log and pid file directories (Issue #545).

  Contributed by Chris Streeter.

9.5 2.4.3

release-date 2011-11-22 18:00 P.M GMT

by Ask Solem

- Fixes module import typo in *celeryctl* (Issue #538).

  Fix contributed by Chris Streeter.

9.6 2.4.2

release-date 2011-11-14 12:00 P.M GMT

by Ask Solem

- Program module no longer uses relative imports so that it is possible to do `python -m celery.bin.name`.

9.7 2.4.1

release-date 2011-11-07 06:00 P.M GMT

by Ask Solem

- celeryctl inspect commands was missing output.
- processes pool: Decrease polling interval for less idle CPU usage.
- processes pool: MaybeEncodingError was not wrapped in ExceptionInfo (Issue #524).
- celeryd: would silence errors occuring after task consumer started.
- logging: Fixed a bug where unicode in stdout redirected log messages couldn't be written (Issue #522).

9.8 2.4.0

release-date 2011-11-04 04:00 P.M GMT

by Ask Solem

9.8.1 Important Notes

- Now supports Python 3.
- Fixed deadlock in worker process handling (Issue #496).

  A deadlock could occur after spawning new child processes because the logging library's mutex was not properly reset after fork.

  The symptoms of this bug affecting would be that the worker simply stops processing tasks, as none of the workers child processes are functioning. There was a greater chance of this bug occurring with `maxtasksperchild` or a time-limit enabled.

  This is a workaround for http://bugs.python.org/issue6721#msg140215.

  Be aware that while this fixes the logging library lock, there could still be other locks initialized in the parent process, introduced by custom code.

  Fix contributed by Harm Verhagen.

- AMQP Result backend: Now expires results by default.

  The default expiration value is now taken from the `CELERY_TASK_RESULT_EXPIRES` setting.

  The old `CELERY_AMQP_TASK_RESULT_EXPIRES` setting has been deprecated and will be removed in version 3.0.

  Note that this means that the result backend requires RabbitMQ 1.1.0 or higher, and that you have to disable expiration if you are running with an older version. You can do so by disabling the `CELERY_TASK_RESULT_EXPIRES` setting:

  ```
  CELERY_TASK_RESULT_EXPIRES = None
  ```

- Eventlet: Fixed problem with shutdown (Issue #457).
- Broker transports can be now be specified using URLs

  The broker can now be specified as an URL instead. This URL must have the format:

  ```
  transport://user:password@hostname:port/virtual_host
  ```

  for example the default broker is written as:

  ```
  amqp://guest:guest@localhost:5672//
  ```

  The scheme is required, so that the host is identified as an URL and not just a host name. User, password, port and virtual_host are optional and defaults to the particular transports default value.

  Note: Note that the path component (virtual_host) always starts with a forward-slash. This is necessary to distinguish between the virtual host `''` (empty) and `'/'`, which are both acceptable virtual host names.

  A virtual host of `'/'` becomes:

  amqp://guest:guest@localhost:5672//

  and a virtual host of `''` (empty) becomes:

  ```
  amqp://guest:guest@localhost:5672/
  ```

  So the leading slash in the path component is **always required**.

In addition the BROKER_URL setting has been added as an alias to BROKER_HOST. Any broker setting specified in both the URL and in the configuration will be ignored, if a setting is not provided in the URL then the value from the configuration will be used as default.

Also, programs now support the *-b|--broker* option to specify a broker URL on the command line:

```
$ celeryd -b redis://localhost

$ celeryctl -b amqp://guest:guest@localhost//e
```

The environment variable CELERY_BROKER_URL can also be used to easily override the default broker used.

- The deprecated `celery.loaders.setup_loader()` function has been removed.
- The CELERY_TASK_ERROR_WHITELIST setting has been replaced by a more flexible approach (Issue #447).

  The error mail sending logic is now available as `Task.ErrorMail`, with the implementation (for reference) in `celery.utils.mail`.

  The error mail class can be sub-classed to gain complete control of when error messages are sent, thus removing the need for a separate white-list setting.

  The CELERY_TASK_ERROR_WHITELIST setting has been deprecated, and will be removed completely in version 3.0.

- Additional Deprecations

  The following functions has been deprecated and is scheduled for removal in version 3.0:

Old function	Alternative
celery.loaders.current_loader	*celery.current_app.loader*
celery.loaders.load_settings	*celery.current_app.conf*
celery.execute.apply	*Task.apply*
celery.execute.apply_async	*Task.apply_async*
celery.execute.delay_task	*celery.execute.send_task*

  The following settings has been deprecated and is scheduled for removal in version 3.0:

Old setting	Alternative
CELERYD_LOG_LEVEL	`celeryd --loglevel=`
CELERYD_LOG_FILE	`celeryd --logfile=`
CELERYBEAT_LOG_LEVEL	`celerybeat --loglevel=`
CELERYBEAT_LOG_FILE	`celerybeat --logfile=`
CELERYMON_LOG_LEVEL	`celerymon --loglevel=`
CELERYMON_LOG_FILE	`celerymon --logfile=`

9.8.2 News

- No longer depends on `pyparsing`.
- Now depends on Kombu 1.4.3.
- CELERY_IMPORTS can now be a scalar value (Issue #485).

  It is too easy to forget to add the comma after the sole element of a tuple, and this is something that often affects newcomers.

The docs should probably use a list in examples, as using a tuple for this doesn't even make sense. Nonetheless, there are many tutorials out there using a tuple, and this change should be a help to new users.

Suggested by jsaxon-cars.

- Fixed a memory leak when using the thread pool (Issue #486).

  Contributed by Kornelijus Survila.

- The statedb was not saved at exit.

  This has now been fixed and it should again remember previously revoked tasks when a `--statedb` is enabled.

- Adds `EMAIL_USE_TLS` to enable secure SMTP connections (Issue #418).

  Contributed by Stefan Kjartansson.

- Now handles missing fields in task messages as documented in the message format documentation.

  - Missing required field throws `InvalidTaskError`
  - Missing args/kwargs is assumed empty.

  Contributed by Chris Chamberlin.

- Fixed race condition in celery.events.state (celerymon/celeryev) where task info would be removed while iterating over it (Issue #501).

- The Cache, Cassandra, MongoDB, Redis and Tyrant backends now respects the `CELERY_RESULT_SERIALIZER` setting (Issue #435).

  This means that only the database (django/sqlalchemy) backends currently does not support using custom serializers.

  Contributed by Steeve Morin

- Logging calls no longer manually formats messages, but delegates that to the logging system, so tools like Sentry can easier work with the messages (Issue #445).

  Contributed by Chris Adams.

- `celeryd_multi` now supports a `stop_verify` command to wait for processes to shutdown.

- Cache backend did not work if the cache key was unicode (Issue #504).

  Fix contributed by Neil Chintomby.

- New setting `CELERY_RESULT_DB_SHORT_LIVED_SESSIONS` added, which if enabled will disable the caching of SQLAlchemy sessions (Issue #449).

  Contributed by Leo Dirac.

- All result backends now implements `__reduce__` so that they can be pickled (Issue #441).

  Fix contributed by Remy Noel

- celeryd-multi did not work on Windows (Issue #472).

- New-style `CELERY_REDIS_*` settings now takes precedence over the old `REDIS_*` configuration keys (Issue #508).

  Fix contributed by Joshua Ginsberg

- Generic celerybeat init script no longer sets *bash -e* (Issue #510).

  Fix contributed by Roger Hu.

- Documented that Chords do not work well with redis-server versions before 2.2.

  Contributed by Dan McGee.

- The `CELERYBEAT_MAX_LOOP_INTERVAL` setting was not respected.
- `inspect.registered_tasks` renamed to `inspect.registered` for naming consistency.

  The previous name is still available as an alias.

  Contributed by Mher Movsisyan

- Worker logged the string representation of args and kwargs without safe guards (Issue #480).
- RHEL init script: Changed celeryd startup priority.

  The default start / stop priorities for MySQL on RHEL are

  ```
  # chkconfig: - 64 36
  ```

  Therefore, if Celery is using a database as a broker / message store, it should be started after the database is up and running, otherwise errors will ensue. This commit changes the priority in the init script to

  ```
  # chkconfig: - 85 15
  ```

  which are the default recommended settings for 3-rd party applications and assure that Celery will be started after the database service & shut down before it terminates.

  Contributed by Yury V. Zaytsev.

- KeyValueStoreBackend.get_many did not respect the `timeout` argument (Issue #512).
- celerybeat/celeryev's –workdir option did not chdir before after configuration was attempted (Issue #506).
- After deprecating 2.4 support we can now name modules correctly, since we can take use of absolute imports.

  Therefore the following internal modules have been renamed:

  celery.concurrency.evlet -> celery.concurrency.eventlet celery.concurrency.evg -> celery.concurrency.gevent

- AUTHORS file is now sorted alphabetically.

  Also, as you may have noticed the contributors of new features/fixes are now mentioned in the Changelog.

9.9 2.3.4

release-date 2011-11-25 16:00 P.M GMT

by Ask Solem

9.9.1 Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/`--gid` arguments to **celeryd-multi**, **celeryd_detach**, **celerybeat** and **celeryev** were used.

  This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

9.9.2 Fixes

- Backported fix for #455 from 2.4 to 2.3.
- Statedb was not saved at shutdown.
- Fixes worker sometimes hanging when hard time limit exceeded.

9.10 2.3.3

release-date 2011-16-09 05:00 P.M BST

by Mher Movsisyan

- Monkey patching `sys.stdout` could result in the worker crashing if the replacing object did not define `isatty()` (Issue #477).
- `CELERYD` option in `/etc/default/celeryd` should not be used with generic init scripts.

9.11 2.3.2

release-date 2011-10-07 05:00 P.M BST

9.11.1 News

- Improved Contributing guide.

  If you'd like to contribute to Celery you should read this guide: http://ask.github.com/celery/contributing.html

  We are looking for contributors at all skill levels, so don't hesitate!
- Now depends on Kombu 1.3.1
- `Task.request` now contains the current worker host name (Issue #460).

  Available as `task.request.hostname`.
- **It is now easier for app subclasses to extend how they are pickled.** (see `celery.app.AppPickler`).

9.11.2 Fixes

- *purge/discard_all* was not working correctly (Issue #455).
- The coloring of log messages didn't handle non-ASCII data well (Issue #427).
- [Windows] the multiprocessing pool tried to import `os.kill` even though this is not available there (Issue #450).
- Fixes case where the worker could become unresponsive because of tasks exceeding the hard time limit.
- The `task-sent` event was missing from the event reference.
- `ResultSet.iterate` now returns results as they finish (Issue #459).

  This was not the case previously, even though the documentation states this was the expected behavior.

- Retries will no longer be performed when tasks are called directly (using `__call__`).

  Instead the exception passed to `retry` will be re-raised.

- Eventlet no longer crashes if autoscale is enabled.

  growing and shrinking eventlet pools is still not supported.

- py24 target removed from `tox.ini`.

9.12 2.3.1

release-date 2011-08-07 08:00 P.M BST

9.12.1 Fixes

- The `CELERY_AMQP_TASK_RESULT_EXPIRES` setting did not work, resulting in an AMQP related error about not being able to serialize floats while trying to publish task states (Issue #446).

9.13 2.3.0

release-date 2011-08-05 12:00 P.M BST

tested cPython: 2.5, 2.6, 2.7; PyPy: 1.5; Jython: 2.5.2

9.13.1 Important Notes

- Now requires Kombu 1.2.1
- Results are now disabled by default.

  The AMQP backend was not a good default because often the users were not consuming the results, resulting in thousands of queues.

  While the queues can be configured to expire if left unused, it was not possible to enable this by default because this was only available in recent RabbitMQ versions (2.1.1+)

  With this change enabling a result backend will be a conscious choice, which will hopefully lead the user to read the documentation and be aware of any common pitfalls with the particular backend.

  The default backend is now a dummy backend (`celery.backends.base.DisabledBackend`). Saving state is simply an noop operation, and AsyncResult.wait(), .result, .state, etc. will raise a `NotImplementedError` telling the user to configure the result backend.

  For help choosing a backend please see *Result Backends*.

  If you depend on the previous default which was the AMQP backend, then you have to set this explicitly before upgrading:

  ```
  CELERY_RESULT_BACKEND = "amqp"
  ```

  Note: For django-celery users the default backend is still `database`, and results are not disabled by default.

- The Debian init scripts have been deprecated in favor of the generic-init.d init scripts.

In addition generic init scripts for celerybeat and celeryev has been added.

9.13.2 News

- Automatic connection pool support.

  The pool is used by everything that requires a broker connection. For example applying tasks, sending broadcast commands, retrieving results with the AMQP result backend, and so on.

  The pool is disabled by default, but you can enable it by configuring the `BROKER_POOL_LIMIT` setting:

  ```
  BROKER_POOL_LIMIT = 10
  ```

  A limit of 10 means a maximum of 10 simultaneous connections can co-exist. Only a single connection will ever be used in a single-thread environment, but in a concurrent environment (threads, greenlets, etc., but not processes) when the limit has been exceeded, any try to acquire a connection will block the thread and wait for a connection to be released. This is something to take into consideration when choosing a limit.

  A limit of `None` or 0 means no limit, and connections will be established and closed every time.

- Introducing Chords (taskset callbacks).

  A chord is a task that only executes after all of the tasks in a taskset has finished executing. It's a fancy term for "taskset callbacks" adopted from Cω).

  It works with all result backends, but the best implementation is currently provided by the Redis result backend.

  Here's an example chord:

  ```
  >>> chord(add.subtask((i, i))
  ...         for i in xrange(100))(tsum.subtask()).get()
  9900
  ```

  Please read the *Chords section in the user guide*, if you want to know more.

- Time limits can now be set for individual tasks.

  To set the soft and hard time limits for a task use the `time_limit` and `soft_time_limit` attributes:

  ```
  import time

  @task(time_limit=60, soft_time_limit=30)
  def sleeptask(seconds):
      time.sleep(seconds)
  ```

  If the attributes are not set, then the workers default time limits will be used.

  New in this version you can also change the time limits for a task at runtime using the `time_limit()` remote control command:

  ```
  >>> from celery.task import control
  >>> control.time_limit("tasks.sleeptask",
  ...                    soft=60, hard=120, reply=True)
  [{'worker1.example.com': {'ok': 'time limits set successfully'}}]
  ```

  Only tasks that starts executing after the time limit change will be affected.

Note: Soft time limits will still not work on Windows or other platforms that do not have the `SIGUSR1` signal.

- **Redis backend configuration directive names changed to include the** `CELERY_` prefix.

Old setting name	Replace with
REDIS_HOST	*CELERY_REDIS_HOST*
REDIS_PORT	*CELERY_REDIS_PORT*
REDIS_DB	*CELERY_REDIS_DB*
REDIS_PASSWORD	*CELERY_REDIS_PASSWORD*

  The old names are still supported but pending deprecation.

- PyPy: The default pool implementation used is now multiprocessing if running on PyPy 1.5.
- celeryd-multi: now supports "pass through" options.

  Pass through options makes it easier to use celery without a configuration file, or just add last-minute options on the command line.

  Example use:

  $ celeryd-multi start 4 -c 2 – broker.host=amqp.example.com broker.vhost=/ celery.disable_rate_limits=yes

- celerybeat: Now retries establishing the connection (Issue #419).
- celeryctl: New `list bindings` command.

  Lists the current or all available bindings, depending on the broker transport used.

- Heartbeat is now sent every 30 seconds (previously every 2 minutes).
- `ResultSet.join_native()` and `iter_native()` is now supported by the Redis and Cache result backends.

  This is an optimized version of `join()` using the underlying backends ability to fetch multiple results at once.

- Can now use SSL when sending error e-mails by enabling the `EMAIL_USE_SSL` setting.
- `events.default_dispatcher()`: Context manager to easily obtain an event dispatcher instance using the connection pool.
- Import errors in the configuration module will not be silenced anymore.
- ResultSet.iterate: Now supports the `timeout`, `propagate` and `interval` arguments.
- `with_default_connection` -> `with default_connection`
- TaskPool.apply_async: Keyword arguments `callbacks` and `errbacks` has been renamed to `callback` and `errback` and take a single scalar value instead of a list.
- No longer propagates errors occurring during process cleanup (Issue #365)
- Added `TaskSetResult.delete()`, which will delete a previously saved taskset result.
- Celerybeat now syncs every 3 minutes instead of only at shutdown (Issue #382).
- Monitors now properly handles unknown events, so user-defined events are displayed.
- Terminating a task on Windows now also terminates all of the tasks child processes (Issue #384).
- celeryd: `-I|--include` option now always searches the current directory to import the specified modules.
- Cassandra backend: Now expires results by using TTLs.

- Functional test suite in `funtests` is now actually working properly, and passing tests.

9.13.3 Fixes

- celeryev was trying to create the pidfile twice.
- celery.contrib.batches: Fixed problem where tasks failed silently (Issue #393).
- Fixed an issue where logging objects would give "<Unrepresentable", even though the objects were.
- `CELERY_TASK_ERROR_WHITE_LIST` is now properly initialized in all loaders.
- celeryd_detach now passes through command-line configuration.
- Remote control command `add_consumer` now does nothing if the queue is already being consumed from.

9.14 2.2.8

release-date 2011-11-25 16:00 P.M GMT

by Ask Solem

9.14.1 Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/`--gid` arguments to **celeryd-multi**, **celeryd_detach**, **celerybeat** and **celeryev** were used.

  This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

9.15 2.2.7

release-date 2011-06-13 16:00 P.M BST

- New signals: `after_setup_logger` and `after_setup_task_logger`

  These signals can be used to augment logging configuration after Celery has set up logging.
- Redis result backend now works with Redis 2.4.4.
- celeryd_multi: The `--gid` option now works correctly.
- celeryd: Retry wrongfully used the repr of the traceback instead of the string representation.
- App.config_from_object: Now loads module, not attribute of module.
- Fixed issue where logging of objects would give "<Unrepresentable: ...>"

9.16 2.2.6

release-date 2011-04-15 16:00 P.M CEST

9.16.1 Important Notes

- Now depends on Kombu 1.1.2.
- Dependency lists now explicitly specifies that we don't want python-dateutil 2.x, as this version only supports py3k.

  If you have installed dateutil 2.0 by accident you should downgrade to the 1.5.0 version:

  ```
  pip install -U python-dateutil==1.5.0
  ```

  or by easy_install:

  ```
  easy_install -U python-dateutil==1.5.0
  ```

9.16.2 Fixes

- The new `WatchedFileHandler` broke Python 2.5 support (Issue #367).
- Task: Don't use `app.main` if the task name is set explicitly.
- Sending emails did not work on Python 2.5, due to a bug in the version detection code (Issue #378).
- Beat: Adds method `ScheduleEntry._default_now`

  This method can be overridden to change the default value of `last_run_at`.
- An error occurring in process cleanup could mask task errors.

  We no longer propagate errors happening at process cleanup, but log them instead. This way they will not interfere with publishing the task result (Issue #365).
- Defining tasks did not work properly when using the Django `shell_plus` utility (Issue #366).
- **`AsyncResult.get` did not accept the `interval` and `propagate`** arguments.
- **celeryd: Fixed a bug where celeryd would not shutdown if a** `socket.error` was raised.

9.17 2.2.5

release-date 2011-03-28 06:00 P.M CEST

9.17.1 Important Notes

- Now depends on Kombu 1.0.7

9.17.2 News

- Our documentation is now hosted by Read The Docs (http://docs.celeryproject.org), and all links have been changed to point to the new URL.
- Logging: Now supports log rotation using external tools like logrotate.d (Issue #321)

  This is accomplished by using the `WatchedFileHandler`, which re-opens the file if it is renamed or deleted.

- ***Using Celery with Redis/Database as the messaging queue.*** **now documents how to configure Redis/Database result** backends.
- gevent: Now supports ETA tasks.

  But gevent still needs `CELERY_DISABLE_RATE_LIMITS=True` to work.
- TaskSet User Guide: now contains TaskSet callback recipes.
- Eventlet: New signals:
  - `eventlet_pool_started`
  - `eventlet_pool_preshutdown`
  - `eventlet_pool_postshutdown`
  - `eventlet_pool_apply`

  See `celery.signals` for more information.
- New `BROKER_TRANSPORT_OPTIONS` setting can be used to pass additional arguments to a particular broker transport.
- celeryd: `worker_pid` is now part of the request info as returned by broadcast commands.
- TaskSet.apply/Taskset.apply_async now accepts an optional `taskset_id` argument.
- The taskset_id (if any) is now available in the Task request context.
- SQLAlchemy result backend: taskset_id and taskset_id columns now have a unique constraint. (Tables need to recreated for this to take affect).
- Task Userguide: Added section about choosing a result backend.
- Removed unused attribute `AsyncResult.uuid`.

9.17.3 Fixes

- multiprocessing.Pool: Fixes race condition when marking job with `WorkerLostError` (Issue #268).

  The process may have published a result before it was terminated, but we have no reliable way to detect that this is the case.

  So we have to wait for 10 seconds before marking the result with WorkerLostError. This gives the result handler a chance to retrieve the result.
- multiprocessing.Pool: Shutdown could hang if rate limits disabled.

  There was a race condition when the MainThread was waiting for the pool semaphore to be released. The ResultHandler now terminates after 5 seconds if there are unacked jobs, but no worker processes left to start them (it needs to timeout because there could still be an ack+result that we haven't consumed from the result queue. It is unlikely we will receive any after 5 seconds with no worker processes).
- celerybeat: Now creates pidfile even if the `--detach` option is not set.
- eventlet/gevent: The broadcast command consumer is now running in a separate greenthread.

  This ensures broadcast commands will take priority even if there are many active tasks.
- Internal module `celery.worker.controllers` renamed to `celery.worker.mediator`.
- celeryd: Threads now terminates the program by calling `os._exit`, as it is the only way to ensure exit in the case of syntax errors, or other unrecoverable errors.

- Fixed typo in `maybe_timedelta` (Issue #352).
- celeryd: Broadcast commands now logs with loglevel debug instead of warning.
- AMQP Result Backend: Now resets cached channel if the connection is lost.
- Polling results with the AMQP result backend was not working properly.
- Rate limits: No longer sleeps if there are no tasks, but rather waits for the task received condition (Performance improvement).
- ConfigurationView: `iter(dict)` should return keys, not items (Issue #362).
- celerybeat: PersistentScheduler now automatically removes a corrupted schedule file (Issue #346).
- Programs that doesn't support positional command line arguments now provides a user friendly error message.
- Programs no longer tries to load the configuration file when showing `--version` (Issue #347).
- Autoscaler: The "all processes busy" log message is now severity debug instead of error.
- celeryd: If the message body can't be decoded, it is now passed through `safe_str` when logging.

  This to ensure we don't get additional decoding errors when trying to log the failure.
- `app.config_from_object`/`app.config_from_envvar` now works for all loaders.
- Now emits a user-friendly error message if the result backend name is unknown (Issue #349).
- `celery.contrib.batches`: Now sets loglevel and logfile in the task request so `task.get_logger` works with batch tasks (Issue #357).
- celeryd: An exception was raised if using the amqp transport and the prefetch count value exceeded 65535 (Issue #359).

  The prefetch count is incremented for every received task with an ETA/countdown defined. The prefetch count is a short, so can only support a maximum value of 65535. If the value exceeds the maximum value we now disable the prefetch count, it is re-enabled as soon as the value is below the limit again.
- cursesmon: Fixed unbound local error (Issue #303).
- eventlet/gevent is now imported on demand so autodoc can import the modules without having eventlet/gevent installed.
- celeryd: Ack callback now properly handles `AttributeError`.
- `Task.after_return` is now always called *after* the result has been written.
- Cassandra Result Backend: Should now work with the latest `pycassa` version.
- multiprocessing.Pool: No longer cares if the putlock semaphore is released too many times. (this can happen if one or more worker processes are killed).
- SQLAlchemy Result Backend: Now returns accidentally removed `date_done` again (Issue #325).
- Task.request contex is now always initialized to ensure calling the task function directly works even if it actively uses the request context.
- Exception occuring when iterating over the result from `TaskSet.apply` fixed.
- eventlet: Now properly schedules tasks with an ETA in the past.

9.18 2.2.4

release-date 2011-02-19 12:00 AM CET

9.18.1 Fixes

- celeryd: 2.2.3 broke error logging, resulting in tracebacks not being logged.
- AMQP result backend: Polling task states did not work properly if there were more than one result message in the queue.
- `TaskSet.apply_async()` and `TaskSet.apply()` now supports an optional `taskset_id` keyword argument (Issue #331).
- The current taskset id (if any) is now available in the task context as `request.taskset` (Issue #329).
- SQLAlchemy result backend: *date_done* was no longer part of the results as it had been accidentally removed. It is now available again (Issue #325).
- SQLAlchemy result backend: Added unique constraint on *Task.task_id* and *TaskSet.taskset_id*. Tables needs to be recreated for this to take effect.
- Fixed exception raised when iterating on the result of `TaskSet.apply()`.
- Tasks Userguide: Added section on choosing a result backend.

9.19 2.2.3

release-date 2011-02-12 04:00 P.M CET

9.19.1 Fixes

- Now depends on Kombu 1.0.3
- Task.retry now supports a `max_retries` argument, used to change the default value.
- *multiprocessing.cpu_count* may raise `NotImplementedError` on platforms where this is not supported (Issue #320).
- Coloring of log messages broke if the logged object was not a string.
- Fixed several typos in the init script documentation.
- A regression caused *Task.exchange* and *Task.routing_key* to no longer have any effect. This is now fixed.
- Routing Userguide: Fixes typo, routers in `CELERY_ROUTES` must be instances, not classes.
- **celeryev** did not create pidfile even though the *`--pidfile`* argument was set.
- Task logger format was no longer used. (Issue #317).

  The id and name of the task is now part of the log message again.
- A safe version of `repr()` is now used in strategic places to ensure objects with a broken `__repr__` does not crash the worker, or otherwise make errors hard to understand (Issue #298).
- Remote control command `active_queues`: did not account for queues added at runtime.

In addition the dictionary replied by this command now has a different structure: the exchange key is now a dictionary containing the exchange declaration in full.

- The `-Q` option to **celeryd** removed unused queue declarations, so routing of tasks could fail.

  Queues are no longer removed, but rather *app.amqp.queues.consume_from()* is used as the list of queues to consume from.

  This ensures all queues are available for routing purposes.

- celeryctl: Now supports the *inspect active_queues* command.

9.20 2.2.2

release-date 2011-02-03 04:00 P.M CET

9.20.1 Fixes

- Celerybeat could not read the schedule properly, so entries in `CELERYBEAT_SCHEDULE` would not be scheduled.
- Task error log message now includes *exc_info* again.
- The *eta* argument can now be used with *task.retry*.

  Previously it was overwritten by the countdown argument.

- celeryd-multi/celeryd_detach: Now logs errors occuring when executing the *celeryd* command.
- daemonizing cookbook: Fixed typo `--time-limit 300` -> `--time-limit=300`
- Colors in logging broke non-string objects in log messages.
- `setup_task_logger` no longer makes assumptions about magic task kwargs.

9.21 2.2.1

release-date 2011-02-02 04:00 P.M CET

9.21.1 Fixes

- Eventlet pool was leaking memory (Issue #308).
- Deprecated function `celery.execute.delay_task` was accidentally removed, now available again.
- `BasePool.on_terminate` stub did not exist
- **celeryd detach: Adds readable error messages if user/group name does not** exist.
- Smarter handling of unicode decod errors when logging errors.

9.22 2.2.0

release-date 2011-02-01 10:00 AM CET

9.22.1 Important Notes

- Carrot has been replaced with Kombu

  Kombu is the next generation messaging framework for Python, fixing several flaws present in Carrot that was hard to fix without breaking backwards compatibility.

  Also it adds:

  - First-class support for virtual transports; Redis, Django ORM, SQLAlchemy, Beanstalk, MongoDB, CouchDB and in-memory.
  - Consistent error handling with introspection,
  - The ability to ensure that an operation is performed by gracefully handling connection and channel errors,
  - Message compression (zlib, bzip2, or custom compression schemes).

  This means that *ghettoq* is no longer needed as the functionality it provided is already available in Celery by default. The virtual transports are also more feature complete with support for exchanges (direct and topic). The Redis transport even supports fanout exchanges so it is able to perform worker remote control commands.

- Magic keyword arguments pending deprecation.

  The magic keyword arguments were responsibile for many problems and quirks: notably issues with tasks and decorators, and name collisions in keyword arguments for the unaware.

  It wasn't easy to find a way to deprecate the magic keyword arguments, but we think this is a solution that makes sense and it will not have any adverse effects for existing code.

  The path to a magic keyword argument free world is:

  - the *celery.decorators* module is deprecated and the decorators can now be found in *celery.task*.
  - The decorators in *celery.task* disables keyword arguments by default
  - All examples in the documentation have been changed to use *celery.task*.

  This means that the following will have magic keyword arguments enabled (old style):

  ```
  from celery.decorators import task

  @task
  def add(x, y, **kwargs):
      print("In task %s" % kwargs["task_id"])
      return x + y
  ```

  And this will not use magic keyword arguments (new style):

  ```
  from celery.task import task

  @task
  def add(x, y):
      print("In task %s" % add.request.id)
      return x + y
  ```

  In addition, tasks can choose not to accept magic keyword arguments by setting the *task.accept_magic_kwargs* attribute.

Deprecation

Using the decorators in `celery.decorators` emits a `PendingDeprecationWarning` with a helpful message urging you to change your code, in version 2.4 this will be replaced with a `DeprecationWarning`, and in version 3.0 the `celery.decorators` module will be removed and no longer exist.

Similarly, the *task.accept_magic_kwargs* attribute will no longer have any effect starting from version 3.0.

- The magic keyword arguments are now available as *task.request*

  This is called *the context*. Using thread-local storage the context contains state that is related to the current request.

  It is mutable and you can add custom attributes that will only be seen by the current task request.

  The following context attributes are always available:

Magic Keyword Argument	Replace with
kwargs["task_id"]	*self.request.id*
kwargs["delivery_info"]	*self.request.delivery_info*
kwargs["task_retries"]	*self.request.retries*
kwargs["logfile"]	*self.request.logfile*
kwargs["loglevel"]	*self.request.loglevel*
kwargs["task_is_eager	*self.request.is_eager*
NEW	*self.request.args*
NEW	*self.request.kwargs*

  In addition, the following methods now automatically uses the current context, so you don't have to pass *kwargs* manually anymore:

  - *task.retry*
  - *task.get_logger*
  - *task.update_state*

- Eventlet support.

  This is great news for I/O-bound tasks!

  To change pool implementations you use the `-P|--pool` argument to **celeryd**, or globally using the `CELERYD_POOL` setting. This can be the full name of a class, or one of the following aliases: *processes*, *eventlet*, *gevent*.

  For more information please see the *Concurrency with Eventlet* section in the User Guide.

  Why not gevent?

  For our first alternative concurrency implementation we have focused on Eventlet, but there is also an experimental gevent pool available. This is missing some features, notably the ability to schedule ETA tasks.

  Hopefully the gevent support will be feature complete by version 2.3, but this depends on user demand (and contributions).

- Python 2.4 support deprecated!

  We're happy^H^H^H^H^Hsad to announce that this is the last version to support Python 2.4.

  You are urged to make some noise if you're currently stuck with Python 2.4. Complain to your package maintainers, sysadmins and bosses: tell them it's time to move on!

Apart from wanting to take advantage of with-statements, coroutines, conditional expressions and enhanced try blocks, the code base now contains so many 2.4 related hacks and workarounds it's no longer just a compromise, but a sacrifice.

If it really isn't your choice, and you don't have the option to upgrade to a newer version of Python, you can just continue to use Celery 2.2. Important fixes can be backported for as long as there is interest.

- *celeryd*: Now supports Autoscaling of child worker processes.

  The *--autoscale* option can be used to configure the minimum and maximum number of child worker processes:

```
--autoscale=AUTOSCALE
     Enable autoscaling by providing
     max_concurrency,min_concurrency.  Example:
       --autoscale=10,3 (always keep 3 processes, but grow to
      10 if necessary).
```

- Remote Debugging of Tasks

  `celery.contrib.rdb` is an extended version of pdb that enables remote debugging of processes that does not have terminal access.

  Example usage:

```
    from celery.contrib import rdb
    from celery.task import task

    @task
    def add(x, y):
        result = x + y
        rdb.set_trace()  # <- set breakpoint
        return result


    :func:`~celery.contrib.rdb.set_trace` sets a breakpoint at the current
    location and creates a socket you can telnet into to remotely debug
    your task.

    The debugger may be started by multiple processes at the same time,
    so rather than using a fixed port the debugger will search for an
    available port, starting from the base port (6900 by default).
    The base port can be changed using the environment variable
    :envvar:`CELERY_RDB_PORT`.

    By default the debugger will only be available from the local host,
    to enable access from the outside you have to set the environment
    variable :envvar:`CELERY_RDB_HOST`.

    When `celeryd` encounters your breakpoint it will log the following
    information::

        [INFO/MainProcess] Got task from broker:
            tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8]
        [WARNING/PoolWorker-1] Remote Debugger:6900:
            Please telnet 127.0.0.1 6900.  Type `exit` in session to continue.
        [2011-01-18 14:25:44,119: WARNING/PoolWorker-1] Remote Debugger:6900:
            Waiting for client...
```

```
If you telnet the port specified you will be presented
with a ``pdb`` shell::

    $ telnet localhost 6900
    Connected to localhost.
    Escape character is '^]'.
    > /opt/devel/demoapp/tasks.py(128)add()
    -> return result
    (Pdb)

Enter ``help`` to get a list of available commands,
It may be a good idea to read the `Python Debugger Manual`_ if
you have never used `pdb` before.
```

- Events are now transient and is using a topic exchange (instead of direct).

  The *CELERYD_EVENT_EXCHANGE*, *CELERYD_EVENT_ROUTING_KEY*, *CELERYD_EVENT_EXCHANGE_TYPE* settings are no longer in use.

  This means events will not be stored until there is a consumer, and the events will be gone as soon as the consumer stops. Also it means there can be multiple monitors running at the same time.

  The routing key of an event is the type of event (e.g. *worker.started*, *worker.heartbeat*, *task.succeeded*, etc. This means a consumer can filter on specific types, to only be alerted of the events it cares about.

  Each consumer will create a unique queue, meaning it is in effect a broadcast exchange.

  This opens up a lot of possibilities, for example the workers could listen for worker events to know what workers are in the neighborhood, and even restart workers when they go down (or use this information to optimize tasks/autoscaling).

  Note: The event exchange has been renamed from "celeryevent" to "celeryev" so it does not collide with older versions.

  If you would like to remove the old exchange you can do so by executing the following command:

  ```
  $ camqadm exchange.delete celeryevent
  ```

- *celeryd* now starts without configuration, and configuration can be specified directly on the command line.

  Configuration options must appear after the last argument, separated by two dashes:

  ```
  $ celeryd -l info -I tasks -- broker.host=localhost broker.vhost=/app
  ```

- Configuration is now an alias to the original configuration, so changes to the original will reflect Celery at runtime.
- *celery.conf* has been deprecated, and modifying *celery.conf.ALWAYS_EAGER* will no longer have any effect.

  The default configuration is now available in the `celery.app.defaults` module. The available configuration options and their types can now be introspected.

- Remote control commands are now provided by *kombu.pidbox*, the generic process mailbox.
- Internal module *celery.worker.listener* has been renamed to *celery.worker.consumer*, and *.CarrotListener* is now *.Consumer*.
- Previously deprecated modules *celery.models* and *celery.management.commands* have now been removed as per the deprecation timeline.

- **[Security: Low severity] Removed *celery.task.RemoteExecuteTask* and** accompanying functions: *dmap*, *dmap_async*, and *execute_remote*.

  Executing arbitrary code using pickle is a potential security issue if someone gains unrestricted access to the message broker.

  If you really need this functionality, then you would have to add this to your own project.

- [Security: Low severity] The *stats* command no longer transmits the broker password.

  One would have needed an authenticated broker connection to receive this password in the first place, but sniffing the password at the wire level would have been possible if using unencrypted communication.

9.22.2 News

- The internal module *celery.task.builtins* has been removed.
- The module *celery.task.schedules* is deprecated, and *celery.schedules* should be used instead.

  For example if you have:

  ```
  from celery.task.schedules import crontab
  ```

  You should replace that with:

  ```
  from celery.schedules import crontab
  ```

  The module needs to be renamed because it must be possible to import schedules without importing the *celery.task* module.

- The following functions have been deprecated and is scheduled for removal in version 2.3:
  - *celery.execute.apply_async*

    Use *task.apply_async()* instead.
  - *celery.execute.apply*

    Use *task.apply()* instead.
  - *celery.execute.delay_task*

    Use *registry.tasks[name].delay()* instead.
- Importing *TaskSet* from *celery.task.base* is now deprecated.

  You should use:

  ```
  >>> from celery.task import TaskSet
  ```

  instead.
- New remote control commands:
  - *active_queues*

    Returns the queue declarations a worker is currently consuming from.
- Added the ability to retry publishing the task message in the event of connection loss or failure.

  This is disabled by default but can be enabled using the `CELERY_TASK_PUBLISH_RETRY` setting, and tweaked by the `CELERY_TASK_PUBLISH_RETRY_POLICY` setting.

  In addition *retry*, and *retry_policy* keyword arguments have been added to *Task.apply_async*.

Note: Using the *retry* argument to *apply_async* requires you to handle the publisher/connection manually.

- Periodic Task classes (*@periodic_task*/*PeriodicTask*) will *not* be deprecated as previously indicated in the source code.

  But you are encouraged to use the more flexible `CELERYBEAT_SCHEDULE` setting.

- Built-in daemonization support of celeryd using *celeryd-multi* is no longer experimental and is considered production quality.

  See *Generic init scripts* if you want to use the new generic init scripts.

- Added support for message compression using the `CELERY_MESSAGE_COMPRESSION` setting, or the *compression* argument to *apply_async*. This can also be set using routers.

- ***celeryd*: Now logs stacktrace of all threads when receiving the** *SIGUSR1* signal. (Does not work on cPython 2.4, Windows or Jython).

  Inspired by https://gist.github.com/737056

- Can now remotely terminate/kill the worker process currently processing a task.

  The *revoke* remote control command now supports a *terminate* argument Default signal is *TERM*, but can be specified using the *signal* argument. Signal can be the uppercase name of any signal defined in the `signal` module in the Python Standard Library.

  Terminating a task also revokes it.

  Example:

```
>>> from celery.task.control import revoke

>>> revoke(task_id, terminate=True)
>>> revoke(task_id, terminate=True, signal="KILL")
>>> revoke(task_id, terminate=True, signal="SIGKILL")
```

- *TaskSetResult.join_native*: Backend-optimized version of *join()*.

  If available, this version uses the backends ability to retrieve multiple results at once, unlike *join()* which fetches the results one by one.

  So far only supported by the AMQP result backend. Support for memcached and Redis may be added later.

- Improved implementations of *TaskSetResult.join* and *AsyncResult.wait*.

  An *interval* keyword argument have been added to both so the polling interval can be specified (default interval is 0.5 seconds).

  A *propagate* keyword argument have been added to *result.wait()*, errors will be returned instead of raised if this is set to False.

  Warning: You should decrease the polling interval when using the database result backend, as frequent polling can result in high database load.

- The PID of the child worker process accepting a task is now sent as a field with the *task-started* event.

- The following fields have been added to all events in the worker class:

  - *sw_ident*: Name of worker software (e.g. celeryd).
  - *sw_ver*: Software version (e.g. 2.2.0).

  - *sw_sys*: Operating System (e.g. Linux, Windows, Darwin).

- For better accuracy the start time reported by the multiprocessing worker process is used when calculating task duration.

  Previously the time reported by the accept callback was used.

- ***celerybeat*: New built-in daemonization support using the *–detach*** option.
- ***celeryev*: New built-in daemonization support using the *–detach*** option.
- *TaskSet.apply_async*: Now supports custom publishers by using the *publisher* argument.
- Added `CELERY_SEND_TASK_SENT_EVENT` setting.

  If enabled an event will be sent with every task, so monitors can track tasks before the workers receive them.

- ***celerybeat*: Now reuses the broker connection when applying** scheduled tasks.
- The configuration module and loader to use can now be specified on the command line.

  For example:

  ```
  $ celeryd --config=celeryconfig.py --loader=myloader.Loader
  ```

- Added signals: *beat_init* and *beat_embedded_init*
  - `celery.signals.beat_init`

    Dispatched when **celerybeat** starts (either standalone or embedded). Sender is the `celery.beat.Service` instance.

  - `celery.signals.beat_embedded_init`

    Dispatched in addition to the `beat_init` signal when **celerybeat** is started as an embedded process. Sender is the `celery.beat.Service` instance.

- Redis result backend: Removed deprecated settings *REDIS_TIMEOUT* and *REDIS_CONNECT_RETRY*.
- CentOS init script for **celeryd** now available in *contrib/centos*.
- Now depends on *pyparsing* version 1.5.0 or higher.

  There have been reported issues using Celery with pyparsing 1.4.x, so please upgrade to the latest version.

- Lots of new unit tests written, now with a total coverage of 95%.

9.22.3 Fixes

- *celeryev* Curses Monitor: Improved resize handling and UI layout (Issue #274 + Issue #276)
- AMQP Backend: Exceptions occurring while sending task results are now propagated instead of silenced.

  celeryd will then show the full traceback of these errors in the log.

- AMQP Backend: No longer deletes the result queue after successful poll, as this should be handled by the `CELERY_AMQP_TASK_RESULT_EXPIRES` setting instead.
- AMQP Backend: Now ensures queues are declared before polling results.
- Windows: celeryd: Show error if running with *-B* option.

  Running celerybeat embedded is known not to work on Windows, so users are encouraged to run celerybeat as a separate service instead.

- Windows: Utilities no longer output ANSI color codes on Windows
- camqadm: Now properly handles Ctrl+C by simply exiting instead of showing confusing traceback.
- Windows: All tests are now passing on Windows.
- Remove bin/ directory, and *scripts* section from setup.py.

  This means we now rely completely on setuptools entrypoints.

9.22.4 Experimental

- Jython: celeryd now runs on Jython using the threaded pool.

  All tests pass, but there may still be bugs lurking around the corners.

- PyPy: celeryd now runs on PyPy.

  It runs without any pool, so to get parallel execution you must start multiple instances (e.g. using **celeryd-multi**).

  Sadly an initial benchmark seems to show a 30% performance decrease on pypy-1.4.1 + JIT. We would like to find out why this is, so stay tuned.

- `PublisherPool`: Experimental pool of task publishers and connections to be used with the *retry* argument to *apply_async*.

  The example code below will re-use connections and channels, and retry sending of the task message if the connection is lost.

```
from celery import current_app

# Global pool
pool = current_app().amqp.PublisherPool(limit=10)

def my_view(request):
    with pool.acquire() as publisher:
        add.apply_async((2, 2), publisher=publisher, retry=True)
```

9.23 2.1.4

release-date 2010-12-03 12:00 P.M CEST

9.23.1 Fixes

- Execution options to *apply_async* now takes precedence over options returned by active routers. This was a regression introduced recently (Issue #244).
- *celeryev* curses monitor: Long arguments are now truncated so curses doesn't crash with out of bounds errors. (Issue #235).
- *celeryd*: Channel errors occurring while handling control commands no longer crash the worker but are instead logged with severity error.
- SQLAlchemy database backend: Fixed a race condition occurring when the client wrote the pending state. Just like the Django database backend, it does no longer save the pending state (Issue #261 + Issue #262).

- Error email body now uses *repr(exception)* instead of *str(exception)*, as the latter could result in Unicode decode errors (Issue #245).
- Error email timeout value is now configurable by using the `EMAIL_TIMEOUT` setting.
- *celeryev*: Now works on Windows (but the curses monitor won't work without having curses).
- Unit test output no longer emits non-standard characters.
- *celeryd*: The broadcast consumer is now closed if the connection is reset.
- *celeryd*: Now properly handles errors occurring while trying to acknowledge the message.
- ***TaskRequest.on_failure* now encodes traceback using the current filesystem** encoding. (Issue #286).
- *EagerResult* can now be pickled (Issue #288).

9.23.2 Documentation

- Adding *Contributing*.
- Added *Optimizing*.
- Added *Security* section to the FAQ.

9.24 2.1.3

release-date 2010-11-09 05:00 P.M CEST

- Fixed deadlocks in *timer2* which could lead to *djcelerymon/celeryev -c* hanging.
- *EventReceiver*: now sends heartbeat request to find workers.

  This means **celeryev** and friends finds workers immediately at startup.
- celeryev cursesmon: Set screen_delay to 10ms, so the screen refreshes more often.
- Fixed pickling errors when pickling `AsyncResult` on older Python versions.
- celeryd: prefetch count was decremented by eta tasks even if there were no active prefetch limits.

9.25 2.1.2

release-data TBA

9.25.1 Fixes

- celeryd: Now sends the *task-retried* event for retried tasks.
- celeryd: Now honors ignore result for `WorkerLostError` and timeout errors.
- celerybeat: Fixed `UnboundLocalError` in celerybeat logging when using logging setup signals.
- celeryd: All log messages now includes *exc_info*.

9.26 2.1.1

release-date 2010-10-14 02:00 P.M CEST

9.26.1 Fixes

- Now working on Windows again.

  Removed dependency on the pwd/grp modules.

- snapshots: Fixed race condition leading to loss of events.
- celeryd: Reject tasks with an eta that cannot be converted to a time stamp.

  See issue #209

- concurrency.processes.pool: The semaphore was released twice for each task (both at ACK and result ready).

  This has been fixed, and it is now released only once per task.

- docs/configuration: Fixed typo *CELERYD_SOFT_TASK_TIME_LIMIT* -> `CELERYD_TASK_SOFT_TIME_LIMIT`.

  See issue #214

- control command *dump_scheduled*: was using old .info attribute
- **celeryd-multi: Fixed *set changed size during iteration* bug** occurring in the restart command.
- celeryd: Accidentally tried to use additional command line arguments.

  This would lead to an error like:

  > *got multiple values for keyword argument 'concurrency'*.
  >
  > Additional command line arguments are now ignored, and does not produce this error. However – we do reserve the right to use positional arguments in the future, so please do not depend on this behavior.

- celerybeat: Now respects routers and task execution options again.
- celerybeat: Now reuses the publisher instead of the connection.
- Cache result backend: Using `float` as the expires argument to *cache.set* is deprecated by the memcached libraries, so we now automatically cast to `int`.
- unit tests: No longer emits logging and warnings in test output.

9.26.2 News

- Now depends on carrot version 0.10.7.
- Added `CELERY_REDIRECT_STDOUTS`, and `CELERYD_REDIRECT_STDOUTS_LEVEL` settings.

  `CELERY_REDIRECT_STDOUTS` is used by **celeryd** and **celerybeat**. All output to *stdout* and *stderr* will be redirected to the current logger if enabled.

  `CELERY_REDIRECT_STDOUTS_LEVEL` decides the log level used and is `WARNING` by default.

- Added `CELERYBEAT_SCHEDULER` setting.

This setting is used to define the default for the -S option to **celerybeat**.

Example:

```
CELERYBEAT_SCHEDULER = "djcelery.schedulers.DatabaseScheduler"
```

- Added Task.expires: Used to set default expiry time for tasks.
- New remote control commands: *add_consumer* and *cancel_consumer*.

  add_consumer(queue, exchange, exchange_type, routing_key, **options)
  Tells the worker to declare and consume from the specified declaration.

  cancel_consumer(*queue_name*)
  Tells the worker to stop consuming from queue (by queue name).

  Commands also added to **celeryctl** and `inspect`.

  Example using celeryctl to start consuming from queue "queue", in exchange "exchange", of type "direct" using binding key "key":

  ```
  $ celeryctl inspect add_consumer queue exchange direct key
  $ celeryctl inspect cancel_consumer queue
  ```

  See *celeryctl: Management Utility* for more information about the **celeryctl** program.

  Another example using `inspect`:

  ```
  >>> from celery.task.control import inspect
  >>> inspect.add_consumer(queue="queue", exchange="exchange",
  ...                      exchange_type="direct",
  ...                      routing_key="key",
  ...                      durable=False,
  ...                      auto_delete=True)

  >>> inspect.cancel_consumer("queue")
  ```

- celerybeat: Now logs the traceback if a message can't be sent.
- celerybeat: Now enables a default socket timeout of 30 seconds.
- README/introduction/homepage: Added link to Flask-Celery.

9.27 2.1.0

release-date 2010-10-08 12:00 P.M CEST

9.27.1 Important Notes

- Celery is now following the versioning semantics defined by semver.

  This means we are no longer allowed to use odd/even versioning semantics By our previous versioning scheme this stable release should have been version 2.2.

- Now depends on Carrot 0.10.7.
- No longer depends on SQLAlchemy, this needs to be installed separately if the database result backend is used.

- django-celery now comes with a monitor for the Django Admin interface. This can also be used if you're not a Django user. See *Django Admin Monitor* and *Using outside of Django* for more information.
- If you get an error after upgrading saying: *AttributeError: 'module' object has no attribute 'system'*,

  Then this is because the *celery.platform* module has been renamed to *celery.platforms* to not collide with the built-in `platform` module.

  You have to remove the old `platform.py` (and maybe `platform.pyc`) file from your previous Celery installation.

  To do this use **python** to find the location of this module:

  ```
  $ python
  >>> import celery.platform
  >>> celery.platform
  <module 'celery.platform' from '/opt/devel/celery/celery/platform.pyc'>
  ```

  Here the compiled module is in `/opt/devel/celery/celery/`, to remove the offending files do:

  ```
  $ rm -f /opt/devel/celery/celery/platform.py*
  ```

9.27.2 News

- Added support for expiration of AMQP results (requires RabbitMQ 2.1.0)

  The new configuration option `CELERY_AMQP_TASK_RESULT_EXPIRES` sets the expiry time in seconds (can be int or float):

  ```
  CELERY_AMQP_TASK_RESULT_EXPIRES = 30 * 60   # 30 minutes.
  CELERY_AMQP_TASK_RESULT_EXPIRES = 0.80      # 800 ms.
  ```

- celeryev: Event Snapshots

  If enabled, **celeryd** sends messages about what the worker is doing. These messages are called "events". The events are used by real-time monitors to show what the cluster is doing, but they are not very useful for monitoring over a longer period of time. Snapshots lets you take "pictures" of the clusters state at regular intervals. This can then be stored in a database to generate statistics with, or even monitoring over longer time periods.

  django-celery now comes with a Celery monitor for the Django Admin interface. To use this you need to run the django-celery snapshot camera, which stores snapshots to the database at configurable intervals. See *Using outside of Django* for information about using this monitor if you're not using Django.

  To use the Django admin monitor you need to do the following:

  1. Create the new database tables.

     $ python manage.py syncdb

  2. Start the django-celery snapshot camera:

     ```
     $ python manage.py celerycam
     ```

  3. Open up the django admin to monitor your cluster.

  The admin interface shows tasks, worker nodes, and even lets you perform some actions, like revoking and rate limiting tasks, and shutting down worker nodes.

There's also a Debian init.d script for `celeryev` available, see *Running celeryd as a daemon* for more information.

New command line arguments to celeryev:

- `-c|--camera`: Snapshot camera class to use.
- `--logfile|-f`: Log file
- `--loglevel|-l`: Log level
- `--maxrate|-r`: Shutter rate limit.
- `--freq|-F`: Shutter frequency

The `--camera` argument is the name of a class used to take snapshots with. It must support the interface defined by `celery.events.snapshot.Polaroid`.

Shutter frequency controls how often the camera thread wakes up, while the rate limit controls how often it will actually take a snapshot. The rate limit can be an integer (snapshots/s), or a rate limit string which has the same syntax as the task rate limit strings (*"200/m"*, *"10/s"*, *"1/h"*, etc).

For the Django camera case, this rate limit can be used to control how often the snapshots are written to the database, and the frequency used to control how often the thread wakes up to check if there's anything new.

The rate limit is off by default, which means it will take a snapshot for every `--frequency` seconds.

See also:

Django Admin Monitor and *Snapshots*.

- `broadcast()`: Added callback argument, this can be used to process replies immediately as they arrive.
- celeryctl: New command-line utility to manage and inspect worker nodes, apply tasks and inspect the results of tasks.

  See also:

  The *celeryctl: Management Utility* section in the *User Guide*.

  Some examples:

  ```
  $ celeryctl apply tasks.add -a '[2, 2]' --countdown=10

  $ celeryctl inspect active
  $ celeryctl inspect registered_tasks
  $ celeryctl inspect scheduled
  $ celeryctl inspect --help
  $ celeryctl apply --help
  ```

- Added the ability to set an expiry date and time for tasks.

  Example:

  ```
  >>> # Task expires after one minute from now.
  >>> task.apply_async(args, kwargs, expires=60)
  >>> # Also supports datetime
  >>> task.apply_async(args, kwargs,
  ...                  expires=datetime.now() + timedelta(days=1)
  ```

  When a worker receives a task that has been expired it will be marked as revoked (`celery.exceptions.TaskRevokedError`).

- Changed the way logging is configured.

  We now configure the root logger instead of only configuring our custom logger. In addition we don't hijack the multiprocessing logger anymore, but instead use a custom logger name for different applications:

Application	Logger Name
celeryd	"celery"
celerybeat	"celery.beat"
celeryev	"celery.ev"

  This means that the *loglevel* and *logfile* arguments will affect all registered loggers (even those from 3rd party libraries). Unless you configure the loggers manually as shown below, that is.

  Users can choose to configure logging by subscribing to the :signal:'~celery.signals.setup_logging' signal:

```
from logging.config import fileConfig
from celery import signals

def setup_logging(**kwargs):
    fileConfig("logging.conf")
signals.setup_logging.connect(setup_logging)
```

  If there are no receivers for this signal, the logging subsystem will be configured using the `--loglevel`/`--logfile` argument, this will be used for *all defined loggers*.

  Remember that celeryd also redirects stdout and stderr to the celery logger, if manually configure logging you also need to redirect the stdouts manually:

```
 from logging.config import fileConfig
 from celery import log

def setup_logging(**kwargs):
     import logging
     fileConfig("logging.conf")
     stdouts = logging.getLogger("mystdoutslogger")
     log.redirect_stdouts_to_logger(stdouts, loglevel=logging.WARNING)
```

- celeryd: Added command-line option `-I`/`--include`:

  A comma separated list of (task) modules to be imported.

  Example:

```
$ celeryd -I app1.tasks,app2.tasks
```

- celeryd: now emits a warning if running as the root user (euid is 0).
- `celery.messaging.establish_connection()`: Ability to override defaults used using keyword argument "defaults".
- celeryd: Now uses *multiprocessing.freeze_support()* so that it should work with **py2exe**, **PyInstaller**, **cx_Freeze**, etc.
- celeryd: Now includes more metadata for the `STARTED` state: PID and host name of the worker that started the task.

  See issue #181

- subtask: Merge additional keyword arguments to *subtask()* into task keyword arguments.

  e.g.:

```
>>> s = subtask((1, 2), {"foo": "bar"}, baz=1)
>>> s.args
(1, 2)
>>> s.kwargs
{"foo": "bar", "baz": 1}
```

See issue #182.

- celeryd: Now emits a warning if there is already a worker node using the same name running on the same virtual host.
- AMQP result backend: Sending of results are now retried if the connection is down.
- **AMQP result backend: *result.get()*: Wait for next state if state is not** in `READY_STATES`.
- TaskSetResult now supports subscription.

```
>>> res = TaskSet(tasks).apply_async()
>>> res[0].get()
```

- Added *Task.send_error_emails* + *Task.error_whitelist*, so these can be configured per task instead of just by the global setting.
- Added *Task.store_errors_even_if_ignored*, so it can be changed per Task, not just by the global setting.
- The crontab scheduler no longer wakes up every second, but implements *remaining_estimate* (*Optimization*).
- **celeryd: Store `FAILURE` result if the** `WorkerLostError` exception occurs (worker process disappeared).
- celeryd: Store `FAILURE` result if one of the **TimeLimitExceeded* exceptions occurs.
- Refactored the periodic task responsible for cleaning up results.
  - **The backend cleanup task is now only added to the schedule if** `CELERY_TASK_RESULT_EXPIRES` is set.
  - If the schedule already contains a periodic task named "celery.backend_cleanup" it won't change it, so the behavior of the backend cleanup task can be easily changed.
  - The task is now run every day at 4:00 AM, rather than every day since the first time it was run (using crontab schedule instead of *run_every*)
  - **Renamed *celery.task.builtins.DeleteExpiredTaskMetaTask* ->** `celery.task.builtins.backend_cleanup`
  - The task itself has been renamed from "celery.delete_expired_task_meta" to "celery.backend_cleanup"

  See issue #134.
- Implemented *AsyncResult.forget* for sqla/cache/redis/tyrant backends. (Forget and remove task result).

  See issue #184.
- `TaskSetResult.join`: Added 'propagate=True' argument.

  When set to `False` exceptions occurring in subtasks will not be re-raised.
- Added *Task.update_state(task_id, state, meta)* as a shortcut to *task.backend.store_result(task_id, meta, state)*.

  The backend interface is "private" and the terminology outdated, so better to move this to `Task` so it can be used.
- timer2: Set *self.running=False* in `stop()` so it won't try to join again on subsequent calls to *stop()*.
- Log colors are now disabled by default on Windows.

- *celery.platform* renamed to `celery.platforms`, so it doesn't collide with the built-in `platform` module.
- Exceptions occurring in Mediator+Pool callbacks are now caught and logged instead of taking down the worker.
- Redis result backend: Now supports result expiration using the Redis *EXPIRE* command.
- unit tests: Don't leave threads running at tear down.
- celeryd: Task results shown in logs are now truncated to 46 chars.
- ***Task.__name__* is now an alias to *self.__class__.__name__*.** This way tasks introspects more like regular functions.
- *Task.retry*: Now raises `TypeError` if kwargs argument is empty.

  See issue #164.
- timedelta_seconds: Use *timedelta.total_seconds* if running on Python 2.7
- `TokenBucket`: Generic Token Bucket algorithm
- `celery.events.state`: Recording of cluster state can now be paused and resumed, including support for buffering.

  `State.`**`freeze`**`(`*buffer=True*`)`
  Pauses recording of the stream.

  If *buffer* is true, events received while being frozen will be buffered, and may be replayed later.

  `State.`**`thaw`**`(`*replay=True*`)`
  Resumes recording of the stream.

  If *replay* is true, then the recorded buffer will be applied.

  `State.`**`freeze_while`**`(`*fun*`)`
  With a function to apply, freezes the stream before, and replays the buffer after the function returns.
- `EventReceiver.capture` Now supports a timeout keyword argument.
- celeryd: The mediator thread is now disabled if `CELERY_RATE_LIMITS` is enabled, and tasks are directly sent to the pool without going through the ready queue (*Optimization*).

9.27.3 Fixes

- Pool: Process timed out by *TimeoutHandler* must be joined by the Supervisor, so don't remove it from the internal process list.

  See issue #192.
- *TaskPublisher.delay_task* now supports exchange argument, so exchange can be overridden when sending tasks in bulk using the same publisher

  See issue #187.
- celeryd no longer marks tasks as revoked if `CELERY_IGNORE_RESULT` is enabled.

  See issue #207.
- AMQP Result backend: Fixed bug with *result.get()* if `CELERY_TRACK_STARTED` enabled.

  result.get() would stop consuming after receiving the `STARTED` state.
- Fixed bug where new processes created by the pool supervisor becomes stuck while reading from the task Queue.

See http://bugs.python.org/issue10037

- Fixed timing issue when declaring the remote control command reply queue

  This issue could result in replies being lost, but have now been fixed.

- Backward compatible *LoggerAdapter* implementation: Now works for Python 2.4.

  Also added support for several new methods: *fatal*, *makeRecord*, *_log*, *log*, *isEnabledFor*, *addHandler*, *removeHandler*.

9.27.4 Experimental

- celeryd-multi: Added daemonization support.

  celeryd-multi can now be used to start, stop and restart worker nodes.

  $ celeryd-multi start jerry elaine george kramer

  This also creates PID files and log files (`celeryd@jerry.pid`, ..., `celeryd@jerry.log`. To specify a location for these files use the *–pidfile* and *–logfile* arguments with the *%n* format:

  ```
  $ celeryd-multi start jerry elaine george kramer \
                  --logfile=/var/log/celeryd@%n.log \
                  --pidfile=/var/run/celeryd@%n.pid
  ```

  Stopping:

  ```
  $ celeryd-multi stop jerry elaine george kramer
  ```

  Restarting. The nodes will be restarted one by one as the old ones are shutdown:

  ```
  $ celeryd-multi restart jerry elaine george kramer
  ```

  Killing the nodes (**WARNING**: Will discard currently executing tasks):

  ```
  $ celeryd-multi kill jerry elaine george kramer
  ```

  See *celeryd-multi help* for help.

- celeryd-multi: *start* command renamed to *show*.

  celeryd-multi start will now actually start and detach worker nodes. To just generate the commands you have to use *celeryd-multi show*.

- celeryd: Added *–pidfile* argument.

  The worker will write its pid when it starts. The worker will not be started if this file exists and the pid contained is still alive.

- Added generic init.d script using *celeryd-multi*

  http://github.com/ask/celery/tree/master/contrib/generic-init.d/celeryd

9.27.5 Documentation

- Added User guide section: Monitoring
- Added user guide section: Periodic Tasks

  Moved from *getting-started/periodic-tasks* and updated.

- tutorials/external moved to new section: “community”.

- References has been added to all sections in the documentation.

  This makes it easier to link between documents.

9.28 2.0.3

release-date 2010-08-27 12:00 P.M CEST

9.28.1 Fixes

- celeryd: Properly handle connection errors happening while closing consumers.
- celeryd: Events are now buffered if the connection is down, then sent when the connection is re-established.
- No longer depends on the `mailer` package.

  This package had a name space collision with *django-mailer*, so its functionality was replaced.
- Redis result backend: Documentation typos: Redis doesn't have database names, but database numbers. The default database is now 0.
- `inspect`: *registered_tasks* was requesting an invalid command because of a typo.

  See issue #170.
- `CELERY_ROUTES`: Values defined in the route should now have precedence over values defined in `CELERY_QUEUES` when merging the two.

  With the follow settings:

  ```
  CELERY_QUEUES = {"cpubound": {"exchange": "cpubound",
                                "routing_key": "cpubound"}}

  CELERY_ROUTES = {"tasks.add": {"queue": "cpubound",
                                 "routing_key": "tasks.add",
                                 "serializer": "json"}}
  ```

  The final routing options for *tasks.add* will become:

  ```
  {"exchange": "cpubound",
   "routing_key": "tasks.add",
   "serializer": "json"}
  ```

  This was not the case before: the values in `CELERY_QUEUES` would take precedence.
- Worker crashed if the value of `CELERY_TASK_ERROR_WHITELIST` was not an iterable
- `apply()`: Make sure *kwargs["task_id"]* is always set.
- *AsyncResult.traceback*: Now returns `None`, instead of raising `KeyError` if traceback is missing.
- `inspect`: Replies did not work correctly if no destination was specified.
- Can now store result/metadata for custom states.
- celeryd: A warning is now emitted if the sending of task error emails fails.
- celeryev: Curses monitor no longer crashes if the terminal window is resized.

  See issue #160.
- celeryd: On OS X it is not possible to run *os.exec** in a process that is threaded.

This breaks the SIGHUP restart handler, and is now disabled on OS X, emitting a warning instead.

See issue #152.

- `celery.execute.trace`: Properly handle *raise(str)*, which is still allowed in Python 2.4.

  See issue #175.

- Using urllib2 in a periodic task on OS X crashed because of the proxy auto detection used in OS X.

  This is now fixed by using a workaround. See issue #143.

- Debian init scripts: Commands should not run in a sub shell

  See issue #163.

- Debian init scripts: Use the absolute path of celeryd to allow stat

  See issue #162.

9.28.2 Documentation

- getting-started/broker-installation: Fixed typo

  set_permissions "" -> set_permissions "."*.

- Tasks User Guide: Added section on database transactions.

  See issue #169.

- Routing User Guide: Fixed typo *"feed": -> {"queue": "feeds"}*.

  See issue #169.

- Documented the default values for the `CELERYD_CONCURRENCY` and `CELERYD_PREFETCH_MULTIPLIER` settings.

- Tasks User Guide: Fixed typos in the subtask example

- celery.signals: Documented worker_process_init.

- Daemonization cookbook: Need to export DJANGO_SETTINGS_MODULE in */etc/default/celeryd*.

- Added some more FAQs from stack overflow

- Daemonization cookbook: Fixed typo *CELERYD_LOGFILE/CELERYD_PIDFILE*

  to *CELERYD_LOG_FILE* / *CELERYD_PID_FILE*

  Also added troubleshooting section for the init scripts.

9.29 2.0.2

release-date 2010-07-22 11:31 A.M CEST

- Routes: When using the dict route syntax, the exchange for a task could disappear making the task unroutable.

  See issue #158.

- Test suite now passing on Python 2.4

- No longer have to type *PYTHONPATH*=. to use celeryconfig in the current directory.

This is accomplished by the default loader ensuring that the current directory is in *sys.path* when loading the config module. *sys.path* is reset to its original state after loading.

Adding the current working directory to *sys.path* without the user knowing may be a security issue, as this means someone can drop a Python module in the users directory that executes arbitrary commands. This was the original reason not to do this, but if done *only when loading the config module*, this means that the behavior will only apply to the modules imported in the config module, which I think is a good compromise (certainly better than just explicitly setting *PYTHONPATH*=. anyway)

- Experimental Cassandra backend added.
- celeryd: SIGHUP handler accidentally propagated to worker pool processes.

  In combination with 7a7c44e39344789f11b5346e9cc8340f5fe4846c this would make each child process start a new celeryd when the terminal window was closed :/

- celeryd: Do not install SIGHUP handler if running from a terminal.

  This fixes the problem where celeryd is launched in the background when closing the terminal.

- celeryd: Now joins threads at shutdown.

  See issue #152.

- Test tear down: Don't use *atexit* but nose's *teardown()* functionality instead.

  See issue #154.

- Debian init script for celeryd: Stop now works correctly.
- Task logger: *warn* method added (synonym for *warning*)
- Can now define a white list of errors to send error emails for.

  Example:

  ```
  CELERY_TASK_ERROR_WHITELIST = ('myapp.MalformedInputError')
  ```

  See issue #153.

- celeryd: Now handles overflow exceptions in *time.mktime* while parsing the ETA field.
- LoggerWrapper: Try to detect loggers logging back to stderr/stdout making an infinite loop.
- Added `celery.task.control.inspect`: Inspects a running worker.

  Examples:

  ```
  # Inspect a single worker
  >>> i = inspect("myworker.example.com")

  # Inspect several workers
  >>> i = inspect(["myworker.example.com", "myworker2.example.com"])

  # Inspect all workers consuming on this vhost.
  >>> i = inspect()

  ### Methods

  # Get currently executing tasks
  >>> i.active()

  # Get currently reserved tasks
  >>> i.reserved()
  ```

```
# Get the current eta schedule
>>> i.scheduled()

# Worker statistics and info
>>> i.stats()

# List of currently revoked tasks
>>> i.revoked()

# List of registered tasks
>>> i.registered_tasks()
```

- Remote control commands *dump_active/dump_reserved/dump_schedule* now replies with detailed task requests.

  Containing the original arguments and fields of the task requested.

  In addition the remote control command *set_loglevel* has been added, this only changes the log level for the main process.

- Worker control command execution now catches errors and returns their string representation in the reply.
- Functional test suite added

  `celery.tests.functional.case` contains utilities to start and stop an embedded celeryd process, for use in functional testing.

9.30 2.0.1

release-date 2010-07-09 03:02 P.M CEST

- multiprocessing.pool: Now handles encoding errors, so that pickling errors doesn't crash the worker processes.
- The remote control command replies was not working with RabbitMQ 1.8.0's stricter equivalence checks.

  If you've already hit this problem you may have to delete the declaration:

  ```
  $ camqadm exchange.delete celerycrq
  ```

  or:

  ```
  $ python manage.py camqadm exchange.delete celerycrq
  ```

- A bug sneaked in the ETA scheduler that made it only able to execute one task per second(!)

  The scheduler sleeps between iterations so it doesn't consume too much CPU. It keeps a list of the scheduled items sorted by time, at each iteration it sleeps for the remaining time of the item with the nearest deadline. If there are no eta tasks it will sleep for a minimum amount of time, one second by default.

  A bug sneaked in here, making it sleep for one second for every task that was scheduled. This has been fixed, so now it should move tasks like hot knife through butter.

  In addition a new setting has been added to control the minimum sleep interval; `CELERYD_ETA_SCHEDULER_PRECISION`. A good value for this would be a float between 0 and 1, depending on the needed precision. A value of 0.8 means that when the ETA of a task is met, it will take at most 0.8 seconds for the task to be moved to the ready queue.

- Pool: Supervisor did not release the semaphore.

  This would lead to a deadlock if all workers terminated prematurely.

- Added Python version trove classifiers: 2.4, 2.5, 2.6 and 2.7
- Tests now passing on Python 2.7.
- Task.__reduce__: Tasks created using the task decorator can now be pickled.
- setup.py: nose added to *tests_require*.
- Pickle should now work with SQLAlchemy 0.5.x
- New homepage design by Jan Henrik Helmers: http://celeryproject.org
- New Sphinx theme by Armin Ronacher: http://docs.celeryproject.org/
- Fixed "pending_xref" errors shown in the HTML rendering of the documentation. Apparently this was caused by new changes in Sphinx 1.0b2.
- Router classes in `CELERY_ROUTES` are now imported lazily.

  Importing a router class in a module that also loads the Celery environment would cause a circular dependency. This is solved by importing it when needed after the environment is set up.
- `CELERY_ROUTES` was broken if set to a single dict.

  This example in the docs should now work again:

  ```
  CELERY_ROUTES = {"feed.tasks.import_feed": "feeds"}
  ```
- *CREATE_MISSING_QUEUES* was not honored by apply_async.
- New remote control command: *stats*

  Dumps information about the worker, like pool process ids, and total number of tasks executed by type.

  Example reply:

  ```
  [{'worker.local':
       'total': {'tasks.sleeptask': 6},
       'pool': {'timeouts': [None, None],
                'processes': [60376, 60377],
                'max-concurrency': 2,
                'max-tasks-per-child': None,
                'put-guarded-by-semaphore': True}}]
  ```
- New remote control command: *dump_active*

  Gives a list of tasks currently being executed by the worker. By default arguments are passed through repr in case there are arguments that is not JSON encodable. If you know the arguments are JSON safe, you can pass the argument *safe=True*.

  Example reply:

  ```
  >>> broadcast("dump_active", arguments={"safe": False}, reply=True)
  [{'worker.local': [
      {'args': '(1,)',
       'time_start': 1278580542.6300001,
       'name': 'tasks.sleeptask',
       'delivery_info': {
           'consumer_tag': '30',
           'routing_key': 'celery',
           'exchange': 'celery'},
       'hostname': 'casper.local',
       'acknowledged': True,
       'kwargs': '{}',
  ```

```
        'id': '802e93e9-e470-47ed-b913-06de8510aca2',
    }
]}]
```

- Added experimental support for persistent revokes.

  Use the *-S*|*–statedb* argument to celeryd to enable it:

  ```
  $ celeryd --statedb=/var/run/celeryd
  ```

  This will use the file: */var/run/celeryd.db*, as the *shelve* module automatically adds the *.db* suffix.

9.31 2.0.0

release-date 2010-07-02 02:30 P.M CEST

9.31.1 Foreword

Celery 2.0 contains backward incompatible changes, the most important being that the Django dependency has been removed so Celery no longer supports Django out of the box, but instead as an add-on package called django-celery.

We're very sorry for breaking backwards compatibility, but there's also many new and exciting features to make up for the time you lose upgrading, so be sure to read the *News* section.

Quite a lot of potential users have been upset about the Django dependency, so maybe this is a chance to get wider adoption by the Python community as well.

Big thanks to all contributors, testers and users!

9.31.2 Upgrading for Django-users

Django integration has been moved to a separate package: django-celery.

- To upgrade you need to install the django-celery module and change:

  ```
  INSTALLED_APPS = "celery"
  ```

  to:

  ```
  INSTALLED_APPS = "djcelery"
  ```

- If you use *mod_wsgi* you need to add the following line to your *.wsgi* file:

  ```
  import os
  os.environ["CELERY_LOADER"] = "django"
  ```

- The following modules has been moved to django-celery:

Module name	Replace with
celery.models	*djcelery.models*
celery.managers	*djcelery.managers*
celery.views	*djcelery.views*
celery.urls	*djcelery.urls*
celery.management	*djcelery.management*
celery.loaders.djangoapp	*djcelery.loaders*
celery.backends.database	*djcelery.backends.database*
celery.backends.cache	*djcelery.backends.cache*

Importing `djcelery` will automatically setup Celery to use Django loader. loader. It does this by setting the `CELERY_LOADER` environment variable to *"django"* (it won't change it if a loader is already set.)

When the Django loader is used, the "database" and "cache" result backend aliases will point to the `djcelery` backends instead of the built-in backends, and configuration will be read from the Django settings.

9.31.3 Upgrading for others

Database result backend

The database result backend is now using SQLAlchemy instead of the Django ORM, see Supported Databases for a table of supported databases.

The *DATABASE_** settings has been replaced by a single setting: `CELERY_RESULT_DBURI`. The value here should be an SQLAlchemy Connection String, some examples include:

```
# sqlite (filename)
CELERY_RESULT_DBURI = "sqlite:///celerydb.sqlite"

# mysql
CELERY_RESULT_DBURI = "mysql://scott:tiger@localhost/foo"

# postgresql
CELERY_RESULT_DBURI = "postgresql://scott:tiger@localhost/mydatabase"

# oracle
CELERY_RESULT_DBURI = "oracle://scott:tiger@127.0.0.1:1521/sidname"
```

See SQLAlchemy Connection Strings for more information about connection strings.

To specify additional SQLAlchemy database engine options you can use the `CELERY_RESULT_ENGINE_OPTIONS` setting:

```
# echo enables verbose logging from SQLAlchemy.
CELERY_RESULT_ENGINE_OPTIONS = {"echo": True}
```

Cache result backend

The cache result backend is no longer using the Django cache framework, but it supports mostly the same configuration syntax:

```
CELERY_CACHE_BACKEND = "memcached://A.example.com:11211;B.example.com"
```

To use the cache backend you must either have the pylibmc or python-memcached library installed, of which the former is regarded as the best choice.

The support backend types are *memcached://* and *memory://*, we haven't felt the need to support any of the other backends provided by Django.

9.31.4 Backward incompatible changes

- Default (python) loader now prints warning on missing *celeryconfig.py* instead of raising `ImportError`.

  celeryd raises `ImproperlyConfigured` if the configuration is not set up. This makes it possible to use *–help* etc., without having a working configuration.

  Also this makes it possible to use the client side of celery without being configured:

```
>>> from carrot.connection import BrokerConnection
>>> conn = BrokerConnection("localhost", "guest", "guest", "/")
>>> from celery.execute import send_task
>>> r = send_task("celery.ping", args=(), kwargs={}, connection=conn)
>>> from celery.backends.amqp import AMQPBackend
>>> r.backend = AMQPBackend(connection=conn)
>>> r.get()
'pong'
```

- The following deprecated settings has been removed (as scheduled by the deprecation timeline):

Setting name	Replace with
CELERY_AMQP_CONSUMER_QUEUES	*CELERY_QUEUES*
CELERY_AMQP_EXCHANGE	*CELERY_DEFAULT_EXCHANGE*
CELERY_AMQP_EXCHANGE_TYPE	*CELERY_DEFAULT_EXCHANGE_TYPE*
CELERY_AMQP_CONSUMER_ROUTING_KEY	*CELERY_QUEUES*
CELERY_AMQP_PUBLISHER_ROUTING_KEY	*CELERY_DEFAULT_ROUTING_KEY*

- The *celery.task.rest* module has been removed, use `celery.task.http` instead (as scheduled by the deprecation timeline).

- It's no longer allowed to skip the class name in loader names. (as scheduled by the deprecation timeline):

  Assuming the implicit *Loader* class name is no longer supported, if you use e.g.:

```
CELERY_LOADER = "myapp.loaders"
```

  You need to include the loader class name, like this:

```
CELERY_LOADER = "myapp.loaders.Loader"
```

- `CELERY_TASK_RESULT_EXPIRES` now defaults to 1 day.

  Previous default setting was to expire in 5 days.

- AMQP backend: Don't use different values for *auto_delete*.

  This bug became visible with RabbitMQ 1.8.0, which no longer allows conflicting declarations for the auto_delete and durable settings.

  If you've already used celery with this backend chances are you have to delete the previous declaration:

```
$ camqadm exchange.delete celeryresults
```

- Now uses pickle instead of cPickle on Python versions <= 2.5

  cPickle is broken in Python <= 2.5.

  It unsafely and incorrectly uses relative instead of absolute imports, so e.g.:

```
exceptions.KeyError
```

becomes:

```
celery.exceptions.KeyError
```

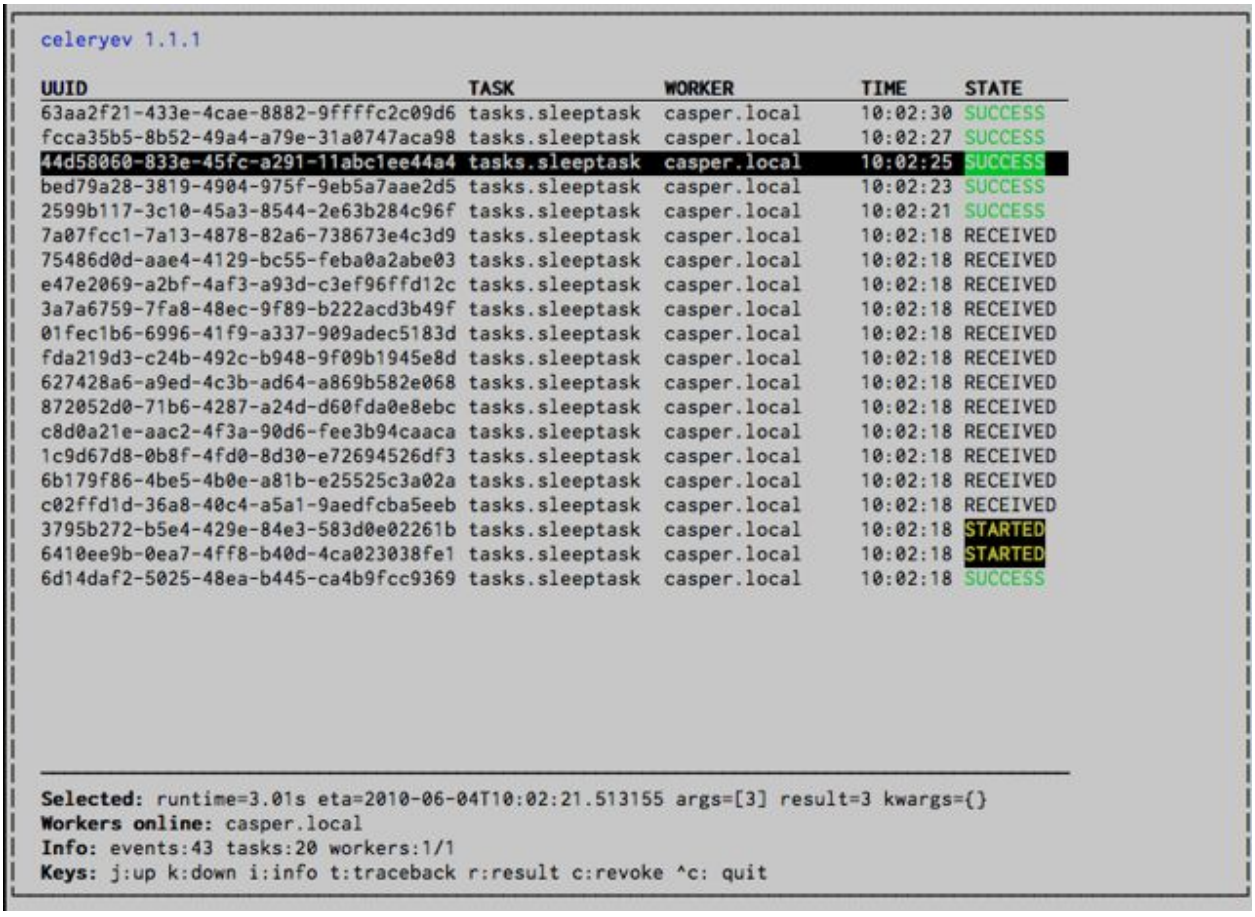
Your best choice is to upgrade to Python 2.6, as while the pure pickle version has worse performance, it is the only safe option for older Python versions.

9.31.5 News

- **celeryev**: Curses Celery Monitor and Event Viewer.

  This is a simple monitor allowing you to see what tasks are executing in real-time and investigate tracebacks and results of ready tasks. It also enables you to set new rate limits and revoke tasks.

  Screenshot:


celeryev 1.1.1
UUID TASK WORKER TIME STATE
63aa2f21-433e-4cae-8882-9ffffc2c09d6 tasks.sleeptask casper.local 10:02:30 SUCCESS
fcca35b5-8b52-49a4-a79e-31a0747aca98 tasks.sleeptask casper.local 10:02:27 SUCCESS
44d58060-833e-45fc-a291-11abc1ee44a4 tasks.sleeptask casper.local 10:02:25 SUCCESS
bed79a28-3819-4904-975f-9eb5a7aae2d5 tasks.sleeptask casper.local 10:02:23 SUCCESS
2599b117-3c10-45a3-8544-2e63b284c96f tasks.sleeptask casper.local 10:02:21 SUCCESS
7a07fcc1-7a13-4878-82a6-738673e4c3d9 tasks.sleeptask casper.local 10:02:18 RECEIVED
75486d0d-aae4-4129-bc55-feba0a2abe03 tasks.sleeptask casper.local 10:02:18 RECEIVED
e47e2069-a2bf-4af3-a93d-c3ef96ffd12c tasks.sleeptask casper.local 10:02:18 RECEIVED
3a7a6759-7fa8-48ec-9f89-b222acd3b49f tasks.sleeptask casper.local 10:02:18 RECEIVED
01fec1b6-6996-41f9-a337-909adec5183d tasks.sleeptask casper.local 10:02:18 RECEIVED
fda219d3-c24b-492c-b948-9f09b1945e8d tasks.sleeptask casper.local 10:02:18 RECEIVED
627428a6-a9ed-4c3b-ad64-a869b582e068 tasks.sleeptask casper.local 10:02:18 RECEIVED
872052d0-71b6-4287-a24d-d60fda0e8ebc tasks.sleeptask casper.local 10:02:18 RECEIVED
c8d0a21e-aac2-4f3a-90d6-fee3b94caaca tasks.sleeptask casper.local 10:02:18 RECEIVED
1c9d67d8-0b8f-4fd0-8d30-e72694526df3 tasks.sleeptask casper.local 10:02:18 RECEIVED
6b179f86-4be5-4b0e-a81b-e25525c3a02a tasks.sleeptask casper.local 10:02:18 RECEIVED
c02ffd1d-36a8-40c4-a5a1-9aedfcba5eeb tasks.sleeptask casper.local 10:02:18 RECEIVED
3795b272-b5e4-429e-84e3-583d0e02261b tasks.sleeptask casper.local 10:02:18 STARTED
6410ee9b-0ea7-4ff8-b40d-4ca023038fe1 tasks.sleeptask casper.local 10:02:18 STARTED
6d14daf2-5025-48ea-b445-ca4b9fcc9369 tasks.sleeptask casper.local 10:02:18 SUCCESS
Selected: runtime=3.01s eta=2010-06-04T10:02:21.513155 args=[3] result=3 kwargs={}
Workers online: casper.local
Info: events:43 tasks:20 workers:1/1
Keys: j:up k:down i:info t:traceback r:result c:revoke ^c: quit

If you run *celeryev* with the *-d* switch it will act as an event dumper, simply dumping the events it receives to standard out:

```
$ celeryev -d
-> celeryev: starting capture...
casper.local [2010-06-04 10:42:07.020000] heartbeat
casper.local [2010-06-04 10:42:14.750000] task received:
```

```
        tasks.add(61a68756-27f4-4879-b816-3cf815672b0e) args=[2, 2] kwargs={}
        eta=2010-06-04T10:42:16.669290, retries=0
    casper.local [2010-06-04 10:42:17.230000] task started
        tasks.add(61a68756-27f4-4879-b816-3cf815672b0e) args=[2, 2] kwargs={}
    casper.local [2010-06-04 10:42:17.960000] task succeeded:
        tasks.add(61a68756-27f4-4879-b816-3cf815672b0e)
        args=[2, 2] kwargs={} result=4, runtime=0.782663106918

    The fields here are, in order: *sender hostname*, *timestamp*, *event type* and
    *additional event fields*.
```

- AMQP result backend: Now supports *.ready()*, *.successful()*, *.result*, *.status*, and even responds to changes in task state
- New user guides:
  - *Workers Guide*
  - *Sets of tasks, Subtasks and Callbacks*
  - *Routing Tasks*
- celeryd: Standard out/error is now being redirected to the log file.
- `billiard` has been moved back to the celery repository.

Module name	celery equivalent
billiard.pool	*celery.concurrency.processes.pool*
billiard.serialization	*celery.serialization*
billiard.utils.functional	*celery.utils.functional*

  The `billiard` distribution may be maintained, depending on interest.
- now depends on `carrot` >= 0.10.5
- now depends on `pyparsing`
- celeryd: Added *–purge* as an alias to *–discard*.
- celeryd: Ctrl+C (SIGINT) once does warm shutdown, hitting Ctrl+C twice forces termination.
- Added support for using complex crontab-expressions in periodic tasks. For example, you can now use:

  ```
  >>> crontab(minute="*/15")
  ```

  or even:

  ```
  >>> crontab(minute="*/30", hour="8-17,1-2", day_of_week="thu-fri")
  ```

  See *Periodic Tasks*.
- celeryd: Now waits for available pool processes before applying new tasks to the pool.

  This means it doesn't have to wait for dozens of tasks to finish at shutdown because it has applied prefetched tasks without having any pool processes available to immediately accept them.

  See issue #122.
- New built-in way to do task callbacks using `subtask`.

  See *Sets of tasks, Subtasks and Callbacks* for more information.
- TaskSets can now contain several types of tasks.

  `TaskSet` has been refactored to use a new syntax, please see *Sets of tasks, Subtasks and Callbacks* for more information.

The previous syntax is still supported, but will be deprecated in version 1.4.

- TaskSet failed() result was incorrect.

  See issue #132.

- Now creates different loggers per task class.

  See issue #129.

- Missing queue definitions are now created automatically.

  You can disable this using the `CELERY_CREATE_MISSING_QUEUES` setting.

  The missing queues are created with the following options:

  ```
  CELERY_QUEUES[name] = {"exchange": name,
                         "exchange_type": "direct",
                         "routing_key": "name}
  ```

  This feature is added for easily setting up routing using the *-Q* option to *celeryd*:

  ```
  $ celeryd -Q video, image
  ```

  See the new routing section of the User Guide for more information: *Routing Tasks*.

- New Task option: *Task.queue*

  If set, message options will be taken from the corresponding entry in `CELERY_QUEUES`. *exchange*, *exchange_type* and *routing_key* will be ignored

- Added support for task soft and hard time limits.

  New settings added:

  - `CELERYD_TASK_TIME_LIMIT`

    Hard time limit. The worker processing the task will be killed and replaced with a new one when this is exceeded.

  - `CELERYD_SOFT_TASK_TIME_LIMIT`

    Soft time limit. The `celery.exceptions.SoftTimeLimitExceeded` exception will be raised when this is exceeded. The task can catch this to e.g. clean up before the hard time limit comes.

  New command line arguments to celeryd added: *–time-limit* and *–soft-time-limit*.

  What's left?

  This won't work on platforms not supporting signals (and specifically the *SIGUSR1* signal) yet. So an alternative the ability to disable the feature all together on nonconforming platforms must be implemented.

  Also when the hard time limit is exceeded, the task result should be a *TimeLimitExceeded* exception.

- Test suite is now passing without a running broker, using the carrot in-memory backend.

- Log output is now available in colors.

Log level	Color
DEBUG	Blue
WARNING	Yellow
CRITICAL	Magenta
ERROR	Red

This is only enabled when the log output is a tty. You can explicitly enable/disable this feature using the CELERYD_LOG_COLOR setting.

- Added support for task router classes (like the django multi-db routers)
  - New setting: CELERY_ROUTES

  This is a single, or a list of routers to traverse when sending tasks. Dictionaries in this list converts to a celery.routes.MapRoute instance.

  Examples:

```
>>> CELERY_ROUTES = {"celery.ping": "default",
                     "mytasks.add": "cpu-bound",
                     "video.encode": {
                         "queue": "video",
                         "exchange": "media"
                         "routing_key": "media.video.encode"}}

>>> CELERY_ROUTES = ("myapp.tasks.Router",
                     {"celery.ping": "default})
```

  Where *myapp.tasks.Router* could be:

```
class Router(object):

    def route_for_task(self, task, args=None, kwargs=None):
        if task == "celery.ping":
            return "default"
```

  route_for_task may return a string or a dict. A string then means it's a queue name in CELERY_QUEUES, a dict means it's a custom route.

  When sending tasks, the routers are consulted in order. The first router that doesn't return *None* is the route to use. The message options is then merged with the found route settings, where the routers settings have priority.

  Example if apply_async() has these arguments:

```
>>> Task.apply_async(immediate=False, exchange="video",
...                  routing_key="video.compress")
```

  and a router returns:

```
{"immediate": True,
 "exchange": "urgent"}
```

  the final message options will be:

```
immediate=True, exchange="urgent", routing_key="video.compress"
```

  (and any default message options defined in the Task class)

- New Task handler called after the task returns: after_return().
- **ExceptionInfo now passed to** on_retry()/on_failure() as einfo keyword argument.
- celeryd: Added CELERYD_MAX_TASKS_PER_CHILD / *--maxtasksperchild*

  Defines the maximum number of tasks a pool worker can process before the process is terminated and replaced by a new one.

- Revoked tasks now marked with state REVOKED, and *result.get()* will now raise TaskRevokedError.

- `celery.task.control.ping()` now works as expected.
- *apply(throw=True)* / `CELERY_EAGER_PROPAGATES_EXCEPTIONS`: Makes eager execution re-raise task errors.
- New signal: `~celery.signals.worker_process_init`: Sent inside the pool worker process at init.
- celeryd `-Q` option: Ability to specify list of queues to use, disabling other configured queues.

  For example, if `CELERY_QUEUES` defines four queues: *image*, *video*, *data* and *default*, the following command would make celeryd only consume from the *image* and *video* queues:

  ```
  $ celeryd -Q image,video
  ```

- celeryd: New return value for the *revoke* control command:

  Now returns:

  ```
  {"ok": "task $id revoked"}
  ```

  instead of *True*.

- celeryd: Can now enable/disable events using remote control

  Example usage:

  ```
  >>> from celery.task.control import broadcast
  >>> broadcast("enable_events")
  >>> broadcast("disable_events")
  ```

- Removed top-level tests directory. Test config now in celery.tests.config

  This means running the unit tests doesn't require any special setup. *celery/tests/__init__* now configures the `CELERY_CONFIG_MODULE` and `CELERY_LOADER` environment variables, so when *nosetests* imports that, the unit test environment is all set up.

  Before you run the tests you need to install the test requirements:

  ```
  $ pip install -r contrib/requirements/test.txt
  ```

  Running all tests:

  ```
  $ nosetests
  ```

  Specifying the tests to run:

  ```
  $ nosetests celery.tests.test_task
  ```

  Producing HTML coverage:

  ```
  $ nosetests --with-coverage3
  ```

  The coverage output is then located in *celery/tests/cover/index.html*.

- celeryd: New option *–version*: Dump version info and exit.
- `celeryd-multi`: Tool for shell scripts to start multiple workers.

  Some examples:

  ```
  # Advanced example with 10 workers:
  #   * Three of the workers processes the images and video queue
  #   * Two of the workers processes the data queue with loglevel DEBUG
  #   * the rest processes the default' queue.
  $ celeryd-multi start 10 -l INFO -Q:1-3 images,video -Q:4,5:data
  ```

```
    -Q default -L:4,5 DEBUG

# get commands to start 10 workers, with 3 processes each
$ celeryd-multi start 3 -c 3
celeryd -n celeryd1.myhost -c 3
celeryd -n celeryd2.myhost -c 3
celeryd- n celeryd3.myhost -c 3

# start 3 named workers
$ celeryd-multi start image video data -c 3
celeryd -n image.myhost -c 3
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3

# specify custom hostname
$ celeryd-multi start 2 -n worker.example.com -c 3
celeryd -n celeryd1.worker.example.com -c 3
celeryd -n celeryd2.worker.example.com -c 3

# Additionl options are added to each celeryd',
# but you can also modify the options for ranges of or single workers

# 3 workers: Two with 3 processes, and one with 10 processes.
$ celeryd-multi start 3 -c 3 -c:1 10
celeryd -n celeryd1.myhost -c 10
celeryd -n celeryd2.myhost -c 3
celeryd -n celeryd3.myhost -c 3

# can also specify options for named workers
$ celeryd-multi start image video data -c 3 -c:image 10
celeryd -n image.myhost -c 10
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3

# ranges and lists of workers in options is also allowed:
# (-c:1-3 can also be written as -c:1,2,3)
$ celeryd-multi start 5 -c 3  -c:1-3 10
celeryd-multi -n celeryd1.myhost -c 10
celeryd-multi -n celeryd2.myhost -c 10
celeryd-multi -n celeryd3.myhost -c 10
celeryd-multi -n celeryd4.myhost -c 3
celeryd-multi -n celeryd5.myhost -c 3

# lists also works with named workers
$ celeryd-multi start foo bar baz xuzzy -c 3 -c:foo,bar,baz 10
celeryd-multi -n foo.myhost -c 10
celeryd-multi -n bar.myhost -c 10
celeryd-multi -n baz.myhost -c 10
celeryd-multi -n xuzzy.myhost -c 3
```

- The worker now calls the result backends *process_cleanup* method *after* task execution instead of before.
- AMQP result backend now supports Pika.

9.32 1.0.6

release-date 2010-06-30 09:57 A.M CEST

- RabbitMQ 1.8.0 has extended their exchange equivalence tests to include *auto_delete* and *durable*. This broke the AMQP backend.

  If you've already used the AMQP backend this means you have to delete the previous definitions:

  ```
  $ camqadm exchange.delete celeryresults
  ```

  or:

  ```
  $ python manage.py camqadm exchange.delete celeryresults
  ```

9.33 1.0.5

release-date 2010-06-01 02:36 P.M CEST

9.33.1 Critical

- SIGINT/Ctrl+C killed the pool, abruptly terminating the currently executing tasks.

  Fixed by making the pool worker processes ignore `SIGINT`.

- Should not close the consumers before the pool is terminated, just cancel the consumers.

  See issue #122.

- Now depends on `billiard` >= 0.3.1
- celeryd: Previously exceptions raised by worker components could stall startup, now it correctly logs the exceptions and shuts down.
- celeryd: Prefetch counts was set too late. QoS is now set as early as possible, so celeryd can't slurp in all the messages at start-up.

9.33.2 Changes

- `celery.contrib.abortable`: Abortable tasks.

  Tasks that defines steps of execution, the task can then be aborted after each step has completed.

- `EventDispatcher`: No longer creates AMQP channel if events are disabled
- Added required RPM package names under *[bdist_rpm]* section, to support building RPMs from the sources using setup.py
- Running unit tests: `NOSE_VERBOSE` environment var now enables verbose output from Nose.
- `celery.execute.apply()`: Pass log file/log level arguments as task kwargs.

  See issue #110.

- celery.execute.apply: Should return exception, not `ExceptionInfo` on error.

  See issue #111.

- Added new entries to the *FAQs*:
  - Should I use retry or acks_late?
  - Can I execute a task by name?

9.34 1.0.4

release-date 2010-05-31 09:54 A.M CEST

- Changelog merged with 1.0.5 as the release was never announced.

9.35 1.0.3

release-date 2010-05-15 03:00 P.M CEST

9.35.1 Important notes

- Messages are now acknowledged *just before* the task function is executed.

  This is the behavior we've wanted all along, but couldn't have because of limitations in the multiprocessing module. The previous behavior was not good, and the situation worsened with the release of 1.0.1, so this change will definitely improve reliability, performance and operations in general.

  For more information please see http://bit.ly/9hom6T

- Database result backend: result now explicitly sets *null=True* as *django-picklefield* version 0.1.5 changed the default behavior right under our noses :(

  See: http://bit.ly/d5OwMr

  This means those who created their celery tables (via syncdb or celeryinit) with picklefield versions >= 0.1.5 has to alter their tables to allow the result field to be *NULL* manually.

  MySQL:

  ```
  ALTER TABLE celery_taskmeta MODIFY result TEXT NULL
  ```

  PostgreSQL:

  ```
  ALTER TABLE celery_taskmeta ALTER COLUMN result DROP NOT NULL
  ```

- Removed *Task.rate_limit_queue_type*, as it was not really useful and made it harder to refactor some parts.
- Now depends on carrot >= 0.10.4
- Now depends on billiard >= 0.3.0

9.35.2 News

- AMQP backend: Added timeout support for *result.get()* / *result.wait()*.
- New task option: *Task.acks_late* (default: `CELERY_ACKS_LATE`)

  Late ack means the task messages will be acknowledged **after** the task has been executed, not *just before*, which is the default behavior.

  Note: This means the tasks may be executed twice if the worker crashes in mid-execution. Not acceptable for most applications, but desirable for others.

- Added crontab-like scheduling to periodic tasks.

Like a cron job, you can specify units of time of when you would like the task to execute. While not a full implementation of cron's features, it should provide a fair degree of common scheduling needs.

You can specify a minute (0-59), an hour (0-23), and/or a day of the week (0-6 where 0 is Sunday, or by names: sun, mon, tue, wed, thu, fri, sat).

Examples:

```
from celery.schedules import crontab
from celery.decorators import periodic_task

@periodic_task(run_every=crontab(hour=7, minute=30))
def every_morning():
    print("Runs every morning at 7:30a.m")

@periodic_task(run_every=crontab(hour=7, minute=30, day_of_week="mon"))
def every_monday_morning():
    print("Run every monday morning at 7:30a.m")

@periodic_task(run_every=crontab(minutes=30))
def every_hour():
    print("Runs every hour on the clock. e.g. 1:30, 2:30, 3:30 etc.")
```

Note: This a late addition. While we have unittests, due to the nature of this feature we haven't been able to completely test this in practice, so consider this experimental.

- *TaskPool.apply_async*: Now supports the *accept_callback* argument.
- *apply_async*: Now raises `ValueError` if task args is not a list, or kwargs is not a tuple (Issue #95).
- *Task.max_retries* can now be *None*, which means it will retry forever.
- Celerybeat: Now reuses the same connection when publishing large sets of tasks.
- Modified the task locking example in the documentation to use *cache.add* for atomic locking.
- Added experimental support for a *started* status on tasks.

  If *Task.track_started* is enabled the task will report its status as "started" when the task is executed by a worker.

  The default value is *False* as the normal behaviour is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried. Having a "started" status can be useful for when there are long running tasks and there is a need to report which task is currently running.

  The global default can be overridden by the `CELERY_TRACK_STARTED` setting.
- User Guide: New section *Tips and Best Practices*.

  Contributions welcome!

9.35.3 Remote control commands

- Remote control commands can now send replies back to the caller.

  Existing commands has been improved to send replies, and the client interface in *celery.task.control* has new keyword arguments: *reply*, *timeout* and *limit*. Where reply means it will wait for replies, timeout is the time in seconds to stop waiting for replies, and limit is the maximum number of replies to get.

  By default, it will wait for as many replies as possible for one second.

– rate_limit(task_name, destination=all, reply=False, timeout=1, limit=0)

Worker returns *{"ok": message}* on success, or *{"failure": message}* on failure.

```
>>> from celery.task.control import rate_limit
>>> rate_limit("tasks.add", "10/s", reply=True)
[{'worker1': {'ok': 'new rate limit set successfully'}},
 {'worker2': {'ok': 'new rate limit set successfully'}}]
```

– ping(destination=all, reply=False, timeout=1, limit=0)

Worker returns the simple message *"pong"*.

```
>>> from celery.task.control import ping
>>> ping(reply=True)
[{'worker1': 'pong'},
 {'worker2': 'pong'},
```

– revoke(destination=all, reply=False, timeout=1, limit=0)

Worker simply returns *True*.

```
>>> from celery.task.control import revoke
>>> revoke("419e46eb-cf6a-4271-86a8-442b7124132c", reply=True)
[{'worker1': True},
 {'worker2'; True}]
```

- You can now add your own remote control commands!

Remote control commands are functions registered in the command registry. Registering a command is done using `celery.worker.control.Panel.register()`:

```
from celery.task.control import Panel

@Panel.register
def reset_broker_connection(panel, **kwargs):
    panel.consumer.reset_connection()
    return {"ok": "connection re-established"}
```

With this module imported in the worker, you can launch the command using *celery.task.control.broadcast*:

```
>>> from celery.task.control import broadcast
>>> broadcast("reset_broker_connection", reply=True)
[{'worker1': {'ok': 'connection re-established'},
 {'worker2': {'ok': 'connection re-established'}}]
```

TIP You can choose the worker(s) to receive the command by using the *destination* argument:

```
>>> broadcast("reset_broker_connection", destination=["worker1"])
[{'worker1': {'ok': 'connection re-established'}]
```

- New remote control command: *dump_reserved*

Dumps tasks reserved by the worker, waiting to be executed:

```
>>> from celery.task.control import broadcast
>>> broadcast("dump_reserved", reply=True)
[{'myworker1': [<TaskRequest ....>]}]
```

- New remote control command: *dump_schedule*

Dumps the workers currently registered ETA schedule. These are tasks with an *eta* (or *countdown*) argument waiting to be executed by the worker.

```
>>> from celery.task.control import broadcast
>>> broadcast("dump_schedule", reply=True)
[{'w1': []},
 {'w3': []},
 {'w2': ['0. 2010-05-12 11:06:00 pri0 <TaskRequest
            {name:"opalfeeds.tasks.refresh_feed_slice",
             id:"95b45760-4e73-4ce8-8eac-f100aa80273a",
             args:"(<Feeds freq_max:3600 freq_min:60
                          start:2184.0 stop:3276.0>,)",
             kwargs:"{'page': 2}"}>']},
 {'w4': ['0. 2010-05-12 11:00:00 pri0 <TaskRequest
            {name:"opalfeeds.tasks.refresh_feed_slice",
             id:"c053480b-58fb-422f-ae68-8d30a464edfe",
             args:"(<Feeds freq_max:3600 freq_min:60
                          start:1092.0 stop:2184.0>,)",
             kwargs:"{\'page\': 1}"}>',
        '1. 2010-05-12 11:12:00 pri0 <TaskRequest
            {name:"opalfeeds.tasks.refresh_feed_slice",
             id:"ab8bc59e-6cf8-44b8-88d0-f1af57789758",
             args:"(<Feeds freq_max:3600 freq_min:60
                          start:3276.0 stop:4365>,)",
             kwargs:"{\'page\': 3}"}>']}]
```

9.35.4 Fixes

- Mediator thread no longer blocks for more than 1 second.

  With rate limits enabled and when there was a lot of remaining time, the mediator thread could block shutdown (and potentially block other jobs from coming in).

- Remote rate limits was not properly applied (Issue #98).
- Now handles exceptions with Unicode messages correctly in *TaskRequest.on_failure*.
- Database backend: *TaskMeta.result*: default value should be *None* not empty string.

9.36 1.0.2

release-date 2010-03-31 12:50 P.M CET

- Deprecated: `CELERY_BACKEND`, please use `CELERY_RESULT_BACKEND` instead.
- We now use a custom logger in tasks. This logger supports task magic keyword arguments in formats.

  The default format for tasks (`CELERYD_TASK_LOG_FORMAT`) now includes the id and the name of tasks so the origin of task log messages can easily be traced.

  Example output::

  [2010-03-25 13:11:20,317: INFO/PoolWorker-1] [tasks.add(a6e1c5ad-60d9-42a0-8b24-9e39363125a4)] Hello from add

  To revert to the previous behavior you can set:

```
CELERYD_TASK_LOG_FORMAT = """
    [%(asctime)s: %(levelname)s/%(processName)s] %(message)s
""".strip()
```

- Unit tests: Don't disable the django test database tear down, instead fixed the underlying issue which was caused by modifications to the *DATABASE_NAME* setting (Issue #82).
- Django Loader: New config `CELERY_DB_REUSE_MAX` (max number of tasks to reuse the same database connection)

  The default is to use a new connection for every task. We would very much like to reuse the connection, but a safe number of reuses is not known, and we don't have any way to handle the errors that might happen, which may even be database dependent.

  See: http://bit.ly/94fwdd
- celeryd: The worker components are now configurable: `CELERYD_POOL`, `CELERYD_CONSUMER`, `CELERYD_MEDIATOR`, and `CELERYD_ETA_SCHEDULER`.

  The default configuration is as follows:

  ```
  CELERYD_POOL = "celery.concurrency.processes.TaskPool"
  CELERYD_MEDIATOR = "celery.worker.controllers.Mediator"
  CELERYD_ETA_SCHEDULER = "celery.worker.controllers.ScheduleController"
  CELERYD_CONSUMER = "celery.worker.consumer.Consumer"
  ```

  The `CELERYD_POOL` setting makes it easy to swap out the multiprocessing pool with a threaded pool, or how about a twisted/eventlet pool?

  Consider the competition for the first pool plug-in started!
- Debian init scripts: Use *-a* not *&&* (Issue #82).
- Debian init scripts: Now always preserves *$CELERYD_OPTS* from the */etc/default/celeryd* and */etc/default/celerybeat*.
- celery.beat.Scheduler: Fixed a bug where the schedule was not properly flushed to disk if the schedule had not been properly initialized.
- celerybeat: Now syncs the schedule to disk when receiving the `SIGTERM` and `SIGINT` signals.
- Control commands: Make sure keywords arguments are not in Unicode.
- ETA scheduler: Was missing a logger object, so the scheduler crashed when trying to log that a task had been revoked.
- management.commands.camqadm: Fixed typo *camqpadm* -> *camqadm* (Issue #83).
- PeriodicTask.delta_resolution: Was not working for days and hours, now fixed by rounding to the nearest day/hour.
- Fixed a potential infinite loop in *BaseAsyncResult.__eq__*, although there is no evidence that it has ever been triggered.
- celeryd: Now handles messages with encoding problems by acking them and emitting an error message.

9.37 1.0.1

release-date 2010-02-24 07:05 P.M CET

- Tasks are now acknowledged early instead of late.

This is done because messages can only be acknowledged within the same connection channel, so if the connection is lost we would have to refetch the message again to acknowledge it.

This might or might not affect you, but mostly those running tasks with a really long execution time are affected, as all tasks that has made it all the way into the pool needs to be executed before the worker can safely terminate (this is at most the number of pool workers, multiplied by the `CELERYD_PREFETCH_MULTIPLIER` setting.)

We multiply the prefetch count by default to increase the performance at times with bursts of tasks with a short execution time. If this doesn't apply to your use case, you should be able to set the prefetch multiplier to zero, without sacrificing performance.

Note: A patch to `multiprocessing` is currently being worked on, this patch would enable us to use a better solution, and is scheduled for inclusion in the *2.0.0* release.

- celeryd now shutdowns cleanly when receiving the `SIGTERM` signal.
- celeryd now does a cold shutdown if the `SIGINT` signal is received (Ctrl+C), this means it tries to terminate as soon as possible.
- Caching of results now moved to the base backend classes, so no need to implement this functionality in the base classes.
- Caches are now also limited in size, so their memory usage doesn't grow out of control.

  You can set the maximum number of results the cache can hold using the `CELERY_MAX_CACHED_RESULTS` setting (the default is five thousand results). In addition, you can refetch already retrieved results using *backend.reload_task_result* + *backend.reload_taskset_result* (that's for those who want to send results incrementally).
- *celeryd* now works on Windows again.

  Warning: If you're using Celery with Django, you can't use *project.settings* as the settings module name, but the following should work:

  ```
  $ python manage.py celeryd --settings=settings
  ```
- Execution: *.messaging.TaskPublisher.send_task* now incorporates all the functionality apply_async previously did.

  Like converting countdowns to eta, so `celery.execute.apply_async()` is now simply a convenient front-end to `celery.messaging.TaskPublisher.send_task()`, using the task classes default options.

  Also `celery.execute.send_task()` has been introduced, which can apply tasks using just the task name (useful if the client does not have the destination task in its task registry).

  Example:

  ```
  >>> from celery.execute import send_task
  >>> result = send_task("celery.ping", args=[], kwargs={})
  >>> result.get()
  'pong'
  ```
- *camqadm*: This is a new utility for command line access to the AMQP API.

  Excellent for deleting queues/bindings/exchanges, experimentation and testing:

  ```
  $ camqadm
  1> help
  ```

Gives an interactive shell, type *help* for a list of commands.

When using Django, use the management command instead:

```
$ python manage.py camqadm
1> help
```

- Redis result backend: To conform to recent Redis API changes, the following settings has been deprecated:
  - *REDIS_TIMEOUT*
  - *REDIS_CONNECT_RETRY*

  These will emit a *DeprecationWarning* if used.

  A *REDIS_PASSWORD* setting has been added, so you can use the new simple authentication mechanism in Redis.
- The redis result backend no longer calls *SAVE* when disconnecting, as this is apparently better handled by Redis itself.
- If *settings.DEBUG* is on, celeryd now warns about the possible memory leak it can result in.
- The ETA scheduler now sleeps at most two seconds between iterations.
- The ETA scheduler now deletes any revoked tasks it might encounter.

  As revokes are not yet persistent, this is done to make sure the task is revoked even though it's currently being hold because its eta is e.g. a week into the future.
- The *task_id* argument is now respected even if the task is executed eagerly (either using apply, or CELERY_ALWAYS_EAGER).
- The internal queues are now cleared if the connection is reset.
- New magic keyword argument: *delivery_info*.

  Used by retry() to resend the task to its original destination using the same exchange/routing_key.
- Events: Fields was not passed by *.send()* (fixes the UUID key errors in celerymon)
- Added *–schedule/-s* option to celeryd, so it is possible to specify a custom schedule filename when using an embedded celerybeat server (the *-B/–beat*) option.
- Better Python 2.4 compatibility. The test suite now passes.
- task decorators: Now preserve docstring as *cls.__doc__*, (was previously copied to *cls.run.__doc__*)
- The *testproj* directory has been renamed to *tests* and we're now using *nose* + *django-nose* for test discovery, and *unittest2* for test cases.
- New pip requirements files available in *contrib/requirements*.
- TaskPublisher: Declarations are now done once (per process).
- Added *Task.delivery_mode* and the CELERY_DEFAULT_DELIVERY_MODE setting.

  These can be used to mark messages non-persistent (i.e. so they are lost if the broker is restarted).
- Now have our own *ImproperlyConfigured* exception, instead of using the Django one.
- Improvements to the Debian init scripts: Shows an error if the program is not executable. Does not modify *CELERYD* when using django with virtualenv.

9.38 1.0.0

release-date 2010-02-10 04:00 P.M CET

9.38.1 Backward incompatible changes

- Celery does not support detaching anymore, so you have to use the tools available on your platform, or something like Supervisord to make celeryd/celerybeat/celerymon into background processes.

  We've had too many problems with celeryd daemonizing itself, so it was decided it has to be removed. Example startup scripts has been added to *contrib/*:

  - Debian, Ubuntu, (start-stop-daemon)

    contrib/debian/init.d/celeryd contrib/debian/init.d/celerybeat

  - Mac OS X launchd

    contrib/mac/org.celeryq.celeryd.plist contrib/mac/org.celeryq.celerybeat.plist contrib/mac/org.celeryq.celerymon.plist

  - Supervisord (http://supervisord.org)

    contrib/supervisord/supervisord.conf

  In addition to *–detach*, the following program arguments has been removed: *–uid*, *–gid*, *–workdir*, *–chroot*, *–pidfile*, *–umask*. All good daemonization tools should support equivalent functionality, so don't worry.

  Also the following configuration keys has been removed: *CELERYD_PID_FILE*, *CELERYBEAT_PID_FILE*, *CELERYMON_PID_FILE*.

- Default celeryd loglevel is now *WARN*, to enable the previous log level start celeryd with *–loglevel=INFO*.

- Tasks are automatically registered.

  This means you no longer have to register your tasks manually. You don't have to change your old code right away, as it doesn't matter if a task is registered twice.

  If you don't want your task to be automatically registered you can set the *abstract* attribute

  ```
  class MyTask(Task):
      abstract = True
  ```

  By using *abstract* only tasks subclassing this task will be automatically registered (this works like the Django ORM).

  If you don't want subclasses to be registered either, you can set the *autoregister* attribute to *False*.

  Incidentally, this change also fixes the problems with automatic name assignment and relative imports. So you also don't have to specify a task name anymore if you use relative imports.

- You can no longer use regular functions as tasks.

  This change was added because it makes the internals a lot more clean and simple. However, you can now turn functions into tasks by using the *@task* decorator:

  ```
  from celery.decorators import task

  @task
  def add(x, y):
      return x + y
  ```

See also:

Tasks for more information about the task decorators.

- The periodic task system has been rewritten to a centralized solution.

  This means *celeryd* no longer schedules periodic tasks by default, but a new daemon has been introduced: *celerybeat*.

  To launch the periodic task scheduler you have to run celerybeat:

  ```
  $ celerybeat
  ```

  Make sure this is running on one server only, if you run it twice, all periodic tasks will also be executed twice.

  If you only have one worker server you can embed it into celeryd like this:

  ```
  $ celeryd --beat # Embed celerybeat in celeryd.
  ```

- The supervisor has been removed.

  This means the *-S* and *–supervised* options to *celeryd* is no longer supported. Please use something like http://supervisord.org instead.

- *TaskSet.join* has been removed, use *TaskSetResult.join* instead.
- The task status *"DONE"* has been renamed to *"SUCCESS"*.
- *AsyncResult.is_done* has been removed, use *AsyncResult.successful* instead.
- The worker no longer stores errors if *Task.ignore_result* is set, to revert to the previous behaviour set `CELERY_STORE_ERRORS_EVEN_IF_IGNORED` to *True*.
- The statistics functionality has been removed in favor of events, so the *-S* and –statistics‘ switches has been removed.
- The module *celery.task.strategy* has been removed.
- *celery.discovery* has been removed, and it's *autodiscover* function is now in *celery.loaders.djangoapp*. Reason: Internal API.
- The `CELERY_LOADER` environment variable now needs loader class name in addition to module name,

  E.g. where you previously had: *"celery.loaders.default"*, you now need *"celery.loaders.default.Loader"*, using the previous syntax will result in a *DeprecationWarning*.

- Detecting the loader is now lazy, and so is not done when importing *celery.loaders*.

  To make this happen *celery.loaders.settings* has been renamed to *load_settings* and is now a function returning the settings object. *celery.loaders.current_loader* is now also a function, returning the current loader.

  So:

  ```
  loader = current_loader
  ```

  needs to be changed to:

  ```
  loader = current_loader()
  ```

9.38.2 Deprecations

- The following configuration variables has been renamed and will be deprecated in v2.0:

  - CELERYD_DAEMON_LOG_FORMAT -> CELERYD_LOG_FORMAT
  - CELERYD_DAEMON_LOG_LEVEL -> CELERYD_LOG_LEVEL
  - CELERY_AMQP_CONNECTION_TIMEOUT -> CELERY_BROKER_CONNECTION_TIMEOUT
  - CELERY_AMQP_CONNECTION_RETRY -> CELERY_BROKER_CONNECTION_RETRY
  - CELERY_AMQP_CONNECTION_MAX_RETRIES -> CELERY_BROKER_CONNECTION_MAX_RETRIES
  - SEND_CELERY_TASK_ERROR_EMAILS -> CELERY_SEND_TASK_ERROR_EMAILS
- The public API names in celery.conf has also changed to a consistent naming scheme.
- We now support consuming from an arbitrary number of queues.

  To do this we had to rename the configuration syntax. If you use any of the custom AMQP routing options (queue/exchange/routing_key, etc.), you should read the new FAQ entry: http://bit.ly/aiWoH.

  The previous syntax is deprecated and scheduled for removal in v2.0.
- *TaskSet.run* has been renamed to *TaskSet.apply_async*.

  TaskSet.run has now been deprecated, and is scheduled for removal in v2.0.

9.38.3 News

- Rate limiting support (per task type, or globally).
- New periodic task system.
- Automatic registration.
- New cool task decorator syntax.
- celeryd now sends events if enabled with the *-E* argument.

  Excellent for monitoring tools, one is already in the making (http://github.com/ask/celerymon).

  Current events include: worker-heartbeat, task-[received/succeeded/failed/retried], worker-online, worker-offline.
- You can now delete (revoke) tasks that has already been applied.
- You can now set the hostname celeryd identifies as using the *–hostname* argument.
- Cache backend now respects the `CELERY_TASK_RESULT_EXPIRES` setting.
- Message format has been standardized and now uses ISO-8601 format for dates instead of datetime.
- *celeryd* now responds to the `SIGHUP` signal by restarting itself.
- Periodic tasks are now scheduled on the clock.

  I.e. *timedelta(hours=1)* means every hour at :00 minutes, not every hour from the server starts. To revert to the previous behaviour you can set *PeriodicTask.relative = True*.
- Now supports passing execute options to a TaskSets list of args, e.g.:

```
>>> ts = TaskSet(add, [([2, 2], {}, {"countdown": 1}),
...                    ([4, 4], {}, {"countdown": 2}),
...                    ([8, 8], {}, {"countdown": 3})])
>>> ts.run()
```

- Got a 3x performance gain by setting the prefetch count to four times the concurrency, (from an average task round-trip of 0.1s to 0.03s!).

A new setting has been added: CELERYD_PREFETCH_MULTIPLIER, which is set to *4* by default.

- Improved support for webhook tasks.

  celery.task.rest is now deprecated, replaced with the new and shiny celery.task.http. With more reflective names, sensible interface, and it's possible to override the methods used to perform HTTP requests.

- The results of task sets are now cached by storing it in the result backend.

9.38.4 Changes

- Now depends on carrot >= 0.8.1
- New dependencies: billiard, python-dateutil, django-picklefield
- No longer depends on python-daemon
- The *uuid* distribution is added as a dependency when running Python 2.4.
- Now remembers the previously detected loader by keeping it in the CELERY_LOADER environment variable.

  This may help on windows where fork emulation is used.

- ETA no longer sends datetime objects, but uses ISO 8601 date format in a string for better compatibility with other platforms.
- No longer sends error mails for retried tasks.
- Task can now override the backend used to store results.
- Refactored the ExecuteWrapper, *apply* and CELERY_ALWAYS_EAGER now also executes the task callbacks and signals.
- Now using a proper scheduler for the tasks with an ETA.

  This means waiting eta tasks are sorted by time, so we don't have to poll the whole list all the time.

- Now also imports modules listed in CELERY_IMPORTS when running with django (as documented).
- Log level for stdout/stderr changed from INFO to ERROR
- ImportErrors are now properly propagated when autodiscovering tasks.
- You can now use *celery.messaging.establish_connection* to establish a connection to the broker.
- When running as a separate service the periodic task scheduler does some smart moves to not poll too regularly.

  If you need faster poll times you can lower the value of CELERYBEAT_MAX_LOOP_INTERVAL.

- You can now change periodic task intervals at runtime, by making *run_every* a property, or subclassing *PeriodicTask.is_due*.
- The worker now supports control commands enabled through the use of a broadcast queue, you can remotely revoke tasks or set the rate limit for a task type. See celery.task.control.
- The services now sets informative process names (as shown in *ps* listings) if the setproctitle module is installed.
- celery.exceptions.NotRegistered now inherits from KeyError, and *TaskRegistry.__getitem__'+'pop* raises *NotRegistered* instead
- You can set the loader via the CELERY_LOADER environment variable.
- You can now set CELERY_IGNORE_RESULT to ignore task results by default (if enabled, tasks doesn't save results or errors to the backend used).

- celeryd now correctly handles malformed messages by throwing away and acknowledging the message, instead of crashing.

9.38.5 Bugs

- Fixed a race condition that could happen while storing task results in the database.

9.38.6 Documentation

- Reference now split into two sections; API reference and internal module reference.

9.39 0.8.4

release-date 2010-02-05 01:52 P.M CEST

- Now emits a warning if the –detach argument is used. –detach should not be used anymore, as it has several not easily fixed bugs related to it. Instead, use something like start-stop-daemon, Supervisord or launchd (os x).
- Make sure logger class is process aware, even if running Python >= 2.6.
- Error emails are not sent anymore when the task is retried.

9.40 0.8.3

release-date 2009-12-22 09:43 A.M CEST

- Fixed a possible race condition that could happen when storing/querying task results using the database backend.
- Now has console script entry points in the setup.py file, so tools like Buildout will correctly install the programs celeryd and celeryinit.

9.41 0.8.2

release-date 2009-11-20 03:40 P.M CEST

- QOS Prefetch count was not applied properly, as it was set for every message received (which apparently behaves like, "receive one more"), instead of only set when our wanted value changed.

9.42 0.8.1

release-date 2009-11-16 05:21 P.M CEST

9.42.1 Very important note

This release (with carrot 0.8.0) enables AMQP QoS (quality of service), which means the workers will only receive as many messages as it can handle at a time. As with any release, you should test this version upgrade on your development servers before rolling it out to production!

9.42.2 Important changes

- If you're using Python < 2.6 and you use the multiprocessing backport, then multiprocessing version 2.6.2.1 is required.
- All AMQP_* settings has been renamed to BROKER_*, and in addition AMQP_SERVER has been renamed to BROKER_HOST, so before where you had:

  ```
  AMQP_SERVER = "localhost"
  AMQP_PORT = 5678
  AMQP_USER = "myuser"
  AMQP_PASSWORD = "mypassword"
  AMQP_VHOST = "celery"
  ```

  You need to change that to:

  ```
  BROKER_HOST = "localhost"
  BROKER_PORT = 5678
  BROKER_USER = "myuser"
  BROKER_PASSWORD = "mypassword"
  BROKER_VHOST = "celery"
  ```

- Custom carrot backends now need to include the backend class name, so before where you had:

  ```
  CARROT_BACKEND = "mycustom.backend.module"
  ```

  you need to change it to:

  ```
  CARROT_BACKEND = "mycustom.backend.module.Backend"
  ```

  where *Backend* is the class name. This is probably *"Backend"*, as that was the previously implied name.
- New version requirement for carrot: 0.8.0

9.42.3 Changes

- Incorporated the multiprocessing backport patch that fixes the *processName* error.
- Ignore the result of PeriodicTask's by default.
- Added a Redis result store backend
- Allow /etc/default/celeryd to define additional options for the celeryd init script.
- MongoDB periodic tasks issue when using different time than UTC fixed.
- Windows specific: Negate test for available os.fork (thanks miracle2k)
- Now tried to handle broken PID files.
- Added a Django test runner to contrib that sets *CELERY_ALWAYS_EAGER = True* for testing with the database backend.
- Added a `CELERY_CACHE_BACKEND` setting for using something other than the django-global cache backend.
- Use custom implementation of functools.partial (curry) for Python 2.4 support (Probably still problems with running on 2.4, but it will eventually be supported)
- Prepare exception to pickle when saving `RETRY` status for all backends.
- SQLite no concurrency limit should only be effective if the database backend is used.

9.43 0.8.0

release-date 2009-09-22 03:06 P.M CEST

9.43.1 Backward incompatible changes

- Add traceback to result value on failure.

  Note: If you use the database backend you have to re-create the database table *celery_taskmeta*.

  Contact the *Mailing list* or *IRC* channel for help doing this.

- Database tables are now only created if the database backend is used, so if you change back to the database backend at some point, be sure to initialize tables (django: *syncdb*, python: *celeryinit*).

  Note: This is only applies if using Django version 1.1 or higher.

- Now depends on *carrot* version 0.6.0.
- Now depends on python-daemon 1.4.8

9.43.2 Important changes

- Celery can now be used in pure Python (outside of a Django project).

  This means celery is no longer Django specific.

  For more information see the FAQ entry *Is Celery for Django only?*.

- Celery now supports task retries.

  See Cookbook: Retrying Tasks for more information.

- We now have an AMQP result store backend.

  It uses messages to publish task return value and status. And it's incredibly fast!

  See issue #6 for more info!

- AMQP QoS (prefetch count) implemented:

  This to not receive more messages than we can handle.

- Now redirects stdout/stderr to the celeryd log file when detached
- **Now uses *inspect.getargspec* to only pass default arguments** the task supports.
- **Add Task.on_success, .on_retry, .on_failure handlers**

  See `celery.task.base.Task.on_success()`, `celery.task.base.Task.on_retry()`, `celery.task.base.Task.on_failure()`,

- ***celery.utils.gen_unique_id*: Workaround for** http://bugs.python.org/issue4607
- **You can now customize what happens at worker start, at process init, etc.,** by creating your own loaders. (see `celery.loaders.default`, `celery.loaders.djangoapp`, `celery.loaders`.)
- Support for multiple AMQP exchanges and queues.

  This feature misses documentation and tests, so anyone interested is encouraged to improve this situation.

- celeryd now survives a restart of the AMQP server!

  Automatically re-establish AMQP broker connection if it's lost.

  New settings:

  - **AMQP_CONNECTION_RETRY** Set to *True* to enable connection retries.
  - **AMQP_CONNECTION_MAX_RETRIES.** Maximum number of restarts before we give up. Default: *100*.

9.43.3 News

- **Fix an incompatibility between python-daemon and multiprocessing,** which resulted in the *[Errno 10] No child processes* problem when detaching.
- **Fixed a possible DjangoUnicodeDecodeError being raised when saving pickled** data to Django's memcached cache backend.
- Better Windows compatibility.
- **New version of the pickled field (taken from** http://www.djangosnippets.org/snippets/513/)
- **New signals introduced: *task_sent*, *task_prerun* and** *task_postrun*, see `celery.signals` for more information.
- ***TaskSetResult.join* caused *TypeError* when *timeout=None*.** Thanks Jerzy Kozera. Closes #31
- ***views.apply* should return *HttpResponse* instance.** Thanks to Jerzy Kozera. Closes #32
- ***PeriodicTask*: Save conversion of *run_every* from *int*** to *timedelta* to the class attribute instead of on the instance.
- **Exceptions has been moved to *celery.exceptions*, but are still** available in the previous module.
- **Try to rollback transaction and retry saving result if an error happens** while setting task status with the database backend.
- jail() refactored into `celery.execute.ExecuteWrapper`.
- *views.apply* now correctly sets mime-type to "application/json"
- *views.task_status* now returns exception if state is `RETRY`
- ***views.task_status* now returns traceback if state is `FAILURE`** or `RETRY`
- Documented default task arguments.
- Add a sensible __repr__ to ExceptionInfo for easier debugging
- **Fix documentation typo *.. import map -> .. import dmap*.** Thanks to mikedizon

9.44 0.6.0

release-date 2009-08-07 06:54 A.M CET

9.44.1 Important changes

- **Fixed a bug where tasks raising unpickleable exceptions crashed pool** workers. So if you've had pool workers mysteriously disappearing, or problems with celeryd stopping working, this has been fixed in this version.

- Fixed a race condition with periodic tasks.
- **The task pool is now supervised, so if a pool worker crashes,** goes away or stops responding, it is automatically replaced with a new one.
- **Task.name is now automatically generated out of class module+name, e.g.** *"djangotwitter.tasks.UpdateStatusesTask"*. Very convenient. No idea why we didn't do this before. Some documentation is updated to not manually specify a task name.

9.44.2 News

- Tested with Django 1.1
- New Tutorial: Creating a click counter using carrot and celery
- **Database entries for periodic tasks are now created at *celeryd*** startup instead of for each check (which has been a forgotten TODO/XXX in the code for a long time)
- **New settings variable: `CELERY_TASK_RESULT_EXPIRES`** Time (in seconds, or a *datetime.timedelta* object) for when after stored task results are deleted. For the moment this only works for the database backend.
- ***celeryd* now emits a debug log message for which periodic tasks** has been launched.
- **The periodic task table is now locked for reading while getting** periodic task status. (MySQL only so far, seeking patches for other engines)
- **A lot more debugging information is now available by turning on the** *DEBUG* log level (*–loglevel=DEBUG*).
- Functions/methods with a timeout argument now works correctly.
- **New: *celery.strategy.even_time_distribution*:** With an iterator yielding task args, kwargs tuples, evenly distribute the processing of its tasks throughout the time window available.
- Log message *Unknown task ignored...* now has log level *ERROR*
- **Log message *"Got task from broker"* is now emitted for all tasks, even if** the task has an ETA (estimated time of arrival). Also the message now includes the ETA for the task (if any).
- **Acknowledgement now happens in the pool callback. Can't do ack in the job** target, as it's not pickleable (can't share AMQP connection, etc.)).
- Added note about .delay hanging in README
- Tests now passing in Django 1.1
- Fixed discovery to make sure app is in INSTALLED_APPS
- **Previously overridden pool behavior (process reap, wait until pool worker** available, etc.) is now handled by *multiprocessing.Pool* itself.
- Convert statistics data to Unicode for use as kwargs. Thanks Lucy!

9.45 0.4.1

release-date 2009-07-02 01:42 P.M CET

- Fixed a bug with parsing the message options (*mandatory*, *routing_key*, *priority*, *immediate*)

9.46 0.4.0

release-date 2009-07-01 07:29 P.M CET

- Adds eager execution. *celery.execute.apply*‘|‘*Task.apply* executes the function blocking until the task is done, for API compatibility it returns an *celery.result.EagerResult* instance. You can configure celery to always run tasks locally by setting the `CELERY_ALWAYS_EAGER` setting to *True*.
- Now depends on *anyjson*.
- 99% coverage using python *coverage* 3.0.

9.47 0.3.20

release-date 2009-06-25 08:42 P.M CET

- New arguments to *apply_async* (the advanced version of *delay_task*), *countdown* and *eta*;

```
>>> # Run 10 seconds into the future.
>>> res = apply_async(MyTask, countdown=10);

>>> # Run 1 day from now
>>> res = apply_async(MyTask,
...                   eta=datetime.now() + timedelta(days=1))
```

- Now unlinks stale PID files
- Lots of more tests.
- Now compatible with carrot >= 0.5.0.
- **IMPORTANT** The *subtask_ids* attribute on the *TaskSetResult* instance has been removed. To get this information instead use:

```
>>> subtask_ids = [subtask.task_id for subtask in ts_res.subtasks]
```

- *Taskset.run()* now respects extra message options from the task class.
- Task: Add attribute *ignore_result*: Don’t store the status and return value. This means you can’t use the *celery.result.AsyncResult* to check if the task is done, or get its return value. Only use if you need the performance and is able live without these features. Any exceptions raised will store the return value/status as usual.
- Task: Add attribute *disable_error_emails* to disable sending error emails for that task.
- Should now work on Windows (although running in the background won’t work, so using the *–detach* argument results in an exception being raised.)
- Added support for statistics for profiling and monitoring. To start sending statistics start *celeryd* with the *–statistics option. Then after a while you can dump the results by running ‘python manage.py celerystats*. See *celery.monitoring* for more information.
- The celery daemon can now be supervised (i.e. it is automatically restarted if it crashes). To use this start celeryd with the –supervised‘ option (or alternatively *-S*).
- views.apply: View applying a task. Example

```
http://e.com/celery/apply/task_name/arg1/arg2//?kwarg1=a&kwarg2=b
```

> **Warning:** Use with caution! Do not expose this URL to the public without first ensuring that your code is safe!

- Refactored *celery.task*. It's now split into three modules:
  - celery.task

    Contains *apply_async*, *delay_task*, *discard_all*, and task shortcuts, plus imports objects from *celery.task.base* and *celery.task.builtins*
  - celery.task.base

    Contains task base classes: *Task*, *PeriodicTask*, *TaskSet*, *AsynchronousMapTask*, *ExecuteRemoteTask*.
  - celery.task.builtins

    Built-in tasks: *PingTask*, *DeleteExpiredTaskMetaTask*.

9.48 0.3.7

release-date 2008-06-16 11:41 P.M CET

- **IMPORTANT** Now uses AMQP's *basic.consume* instead of *basic.get*. This means we're no longer polling the broker for new messages.
- **IMPORTANT** Default concurrency limit is now set to the number of CPUs available on the system.
- **IMPORTANT** *tasks.register*: Renamed *task_name* argument to *name*, so

  ```
  >>> tasks.register(func, task_name="mytask")
  ```

  has to be replaced with:

  ```
  >>> tasks.register(func, name="mytask")
  ```

- The daemon now correctly runs if the pidlock is stale.
- Now compatible with carrot 0.4.5
- Default AMQP connection timeout is now 4 seconds.
- *AsyncResult.read()* was always returning *True*.
- Only use README as long_description if the file exists so easy_install doesn't break.
- *celery.view*: JSON responses now properly set its mime-type.
- *apply_async* now has a *connection* keyword argument so you can re-use the same AMQP connection if you want to execute more than one task.
- Handle failures in task_status view such that it won't throw 500s.
- Fixed typo *AMQP_SERVER* in documentation to *AMQP_HOST*.
- Worker exception emails sent to administrators now works properly.
- No longer depends on *django*, so installing *celery* won't affect the preferred Django version installed.
- Now works with PostgreSQL (psycopg2) again by registering the *PickledObject* field.
- *celeryd*: Added *–detach* option as an alias to *–daemon*, and it's the term used in the documentation from now on.
- Make sure the pool and periodic task worker thread is terminated properly at exit. (So *Ctrl-C* works again).
- Now depends on *python-daemon*.

- Removed dependency to *simplejson*
- Cache Backend: Re-establishes connection for every task process if the Django cache backend is memcached/libmemcached.
- Tyrant Backend: Now re-establishes the connection for every task executed.

9.49 0.3.3

release-date 2009-06-08 01:07 P.M CET

- The *PeriodicWorkController* now sleeps for 1 second between checking for periodic tasks to execute.

9.50 0.3.2

release-date 2009-06-08 01:07 P.M CET

- celeryd: Added option *–discard*: Discard (delete!) all waiting messages in the queue.
- celeryd: The *–wakeup-after* option was not handled as a float.

9.51 0.3.1

release-date 2009-06-08 01:07 P.M CET

- The *PeriodicTask* worker is now running in its own thread instead of blocking the *TaskController* loop.
- Default *QUEUE_WAKEUP_AFTER* has been lowered to *0.1* (was *0.3*)

9.52 0.3.0

release-date 2009-06-08 12:41 P.M CET

Warning: This is a development version, for the stable release, please see versions 0.2.x.

VERY IMPORTANT: Pickle is now the encoder used for serializing task arguments, so be sure to flush your task queue before you upgrade.

- **IMPORTANT** TaskSet.run() now returns a celery.result.TaskSetResult instance, which lets you inspect the status and return values of a taskset as it was a single entity.
- **IMPORTANT** Celery now depends on carrot >= 0.4.1.
- The celery daemon now sends task errors to the registered admin emails. To turn off this feature, set *SEND_CELERY_TASK_ERROR_EMAILS* to *False* in your *settings.py*. Thanks to Grégoire Cachet.
- You can now run the celery daemon by using *manage.py*:

  ```
  $ python manage.py celeryd
  ```

  Thanks to Grégoire Cachet.

- Added support for message priorities, topic exchanges, custom routing keys for tasks. This means we have introduced *celery.task.apply_async*, a new way of executing tasks.

  You can use *celery.task.delay* and *celery.Task.delay* like usual, but if you want greater control over the message sent, you want *celery.task.apply_async* and *celery.Task.apply_async*.

  This also means the AMQP configuration has changed. Some settings has been renamed, while others are new:

  ```
  CELERY_AMQP_EXCHANGE
  CELERY_AMQP_PUBLISHER_ROUTING_KEY
  CELERY_AMQP_CONSUMER_ROUTING_KEY
  CELERY_AMQP_CONSUMER_QUEUE
  CELERY_AMQP_EXCHANGE_TYPE
  ```

  See the entry Can I send some tasks to only some servers? in the FAQ for more information.

- Task errors are now logged using log level *ERROR* instead of *INFO*, and stacktraces are dumped. Thanks to Grégoire Cachet.
- Make every new worker process re-establish it's Django DB connection, this solving the "MySQL connection died?" exceptions. Thanks to Vitaly Babiy and Jirka Vejrazka.
- **IMPORTANT** Now using pickle to encode task arguments. This means you now can pass complex python objects to tasks as arguments.
- Removed dependency to *yadayada*.
- Added a FAQ, see *docs/faq.rst*.
- Now converts any Unicode keys in task *kwargs* to regular strings. Thanks Vitaly Babiy.
- Renamed the *TaskDaemon* to *WorkController*.
- *celery.datastructures.TaskProcessQueue* is now renamed to *celery.pool.TaskPool*.
- The pool algorithm has been refactored for greater performance and stability.

9.53 0.2.0

release-date 2009-05-20 05:14 P.M CET

- Final release of 0.2.0
- Compatible with carrot version 0.4.0.
- Fixes some syntax errors related to fetching results from the database backend.

9.54 0.2.0-pre3

release-date 2009-05-20 05:14 P.M CET

- *Internal release*. Improved handling of unpickleable exceptions, *get_result* now tries to recreate something looking like the original exception.

9.55 0.2.0-pre2

release-date 2009-05-20 01:56 P.M CET

- Now handles unpickleable exceptions (like the dynamically generated subclasses of *django.core.exception.MultipleObjectsReturned*).

9.56 0.2.0-pre1

release-date 2009-05-20 12:33 P.M CET

- It's getting quite stable, with a lot of new features, so bump version to 0.2. This is a pre-release.
- *celery.task.mark_as_read()* and *celery.task.mark_as_failure()* has been removed. Use *celery.backends.default_backend.mark_as_read()*, and *celery.backends.default_backend.mark_as_failure()* instead.

9.57 0.1.15

release-date 2009-05-19 04:13 P.M CET

- The celery daemon was leaking AMQP connections, this should be fixed, if you have any problems with too many files open (like *emfile* errors in *rabbit.log*, please contact us!

9.58 0.1.14

release-date 2009-05-19 01:08 P.M CET

- Fixed a syntax error in the *TaskSet* class. (No such variable *TimeOutError*).

9.59 0.1.13

release-date 2009-05-19 12:36 P.M CET

- Forgot to add *yadayada* to install requirements.
- Now deletes all expired task results, not just those marked as done.
- Able to load the Tokyo Tyrant backend class without django configuration, can specify tyrant settings directly in the class constructor.
- Improved API documentation
- Now using the Sphinx documentation system, you can build the html documentation by doing

```
$ cd docs
$ make html
```

and the result will be in *docs/.build/html*.

9.60 0.1.12

release-date 2009-05-18 04:38 P.M CET

- *delay_task()* etc. now returns *celery.task.AsyncResult* object, which lets you check the result and any failure that might have happened. It kind of works like the *multiprocessing.AsyncResult* class returned by *multiprocessing.Pool.map_async*.
- Added dmap() and dmap_async(). This works like the *multiprocessing.Pool* versions except they are tasks distributed to the celery server. Example:

```
>>> from celery.task import dmap
>>> import operator
>>> dmap(operator.add, [[2, 2], [4, 4], [8, 8]])
>>> [4, 8, 16]

>>> from celery.task import dmap_async
>>> import operator
>>> result = dmap_async(operator.add, [[2, 2], [4, 4], [8, 8]])
>>> result.ready()
False
>>> time.sleep(1)
>>> result.ready()
True
>>> result.result
[4, 8, 16]
```

- Refactored the task metadata cache and database backends, and added a new backend for Tokyo Tyrant. You can set the backend in your django settings file. E.g.:

```
CELERY_RESULT_BACKEND = "database"; # Uses the database
CELERY_RESULT_BACKEND = "cache"; # Uses the django cache framework
CELERY_RESULT_BACKEND = "tyrant"; # Uses Tokyo Tyrant
TT_HOST = "localhost"; # Hostname for the Tokyo Tyrant server.
TT_PORT = 6657; # Port of the Tokyo Tyrant server.
```

9.61 0.1.11

release-date 2009-05-12 02:08 P.M CET

- The logging system was leaking file descriptors, resulting in servers stopping with the EMFILES (too many open files) error. (fixed)

9.62 0.1.10

release-date 2009-05-11 12:46 P.M CET

- Tasks now supports both positional arguments and keyword arguments.
- Requires carrot 0.3.8.
- The daemon now tries to reconnect if the connection is lost.

9.63 0.1.8

release-date 2009-05-07 12:27 P.M CET

- Better test coverage

- More documentation
- celeryd doesn't emit *Queue is empty* message if *settings.CELERYD_EMPTY_MSG_EMIT_EVERY* is 0.

9.64 0.1.7

release-date 2009-04-30 01:50 P.M CET

- Added some unit tests
- Can now use the database for task metadata (like if the task has been executed or not). Set *settings.CELERY_TASK_META*
- Can now run *python setup.py test* to run the unit tests from within the *tests* project.
- Can set the AMQP exchange/routing key/queue using *settings.CELERY_AMQP_EXCHANGE*, *settings.CELERY_AMQP_ROUTING_KEY*, and *settings.CELERY_AMQP_CONSUMER_QUEUE*.

9.65 0.1.6

release-date 2009-04-28 02:13 P.M CET

- Introducing *TaskSet*. A set of subtasks is executed and you can find out how many, or if all them, are done (excellent for progress bars and such)
- Now catches all exceptions when running *Task.__call__*, so the daemon doesn't die. This doesn't happen for pure functions yet, only *Task* classes.
- *autodiscover()* now works with zipped eggs.
- celeryd: Now adds current working directory to *sys.path* for convenience.
- The *run_every* attribute of *PeriodicTask* classes can now be a *datetime.timedelta()* object.
- celeryd: You can now set the *DJANGO_PROJECT_DIR* variable for *celeryd* and it will add that to *sys.path* for easy launching.
- Can now check if a task has been executed or not via HTTP.
- You can do this by including the celery *urls.py* into your project,

```
>>> url(r'^celery/$', include("celery.urls"))
```

then visiting the following url,:

```
http://mysite/celery/$task_id/done/
```

this will return a JSON dictionary like e.g:

```
>>> {"task": {"id": $task_id, "executed": true}}
```

- *delay_task* now returns string id, not *uuid.UUID* instance.
- Now has *PeriodicTasks*, to have *cron* like functionality.
- Project changed name from *crunchy* to *celery*. The details of the name change request is in *docs/name_change_request.txt*.

9.66 0.1.0

release-date 2009-04-24 11:28 A.M CET

- Initial release

CHAPTER 10

API Reference

Release 2.4

Date February 04, 2014

10.1 celery.app

Celery Application.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

- Application
- Functions

10.1.1 Application

class `celery.app.`**App**(*main=None*, *loader=None*, *backend=None*, *amqp=None*, *events=None*, *log=None*, *control=None*, *set_as_current=True*, *accept_magic_kwargs=False*, ***kwargs*)

Celery Application.

Parameters

- **main** – Name of the main module if running as *__main__*.
- **loader** – The loader class, or the name of the loader class to use. Default is `celery.loaders.app.AppLoader`.
- **backend** – The result store backend class, or the name of the backend class to use. Default is the value of the `CELERY_RESULT_BACKEND` setting.
- **amqp** – AMQP object or class name.
- **events** – Events object or class name.
- **log** – Log object or class name.
- **control** – Control object or class name.
- **set_as_current** – Make this the global current app.

main
Name of the *__main__* module. Required for standalone scripts.

If set this will be used instead of *__main__* when automatically generating task names.

amqp
Sending/receiving messages. See `AMQP`.

backend
Storing/retrieving task state. See `BaseBackend`.

loader
Current loader.

conf
Current configuration (dict and attribute access).

control
Controlling worker nodes. See `Control`.

log
Logging utilities. See `Logging`.

config_from_object(*obj*, *silent=False*)
Read configuration from object, where object is either a object, or the name of a module to import.

```
>>> celery.config_from_object("myapp.celeryconfig")

>>> from myapp import celeryconfig
>>> celery.config_from_object(celeryconfig)
```

config_from_envvar(*variable_name*, *silent=False*)
Read configuration from environment variable.

The value of the environment variable must be the name of a module to import.

```
>>> os.environ["CELERY_CONFIG_MODULE"] = "myapp.celeryconfig"
>>> celery.config_from_envvar("CELERY_CONFIG_MODULE")
```

config_from_cmdline(*argv*, *namespace='celery'*)
Read configuration from argv.

The config

task(**args*, ***options*)
Decorator to create a task class out of any callable.

Examples

```
@task()
def refresh_feed(url):
    return Feed.objects.get(url=url).refresh()
```

With setting extra options and using retry.

```
@task(exchange="feeds")
def refresh_feed(url, **kwargs):
    try:
        return Feed.objects.get(url=url).refresh()
    except socket.error, exc:
        refresh_feed.retry(args=[url], kwargs=kwargs, exc=exc)
```

Calling the resulting task:

```
>>> refresh_feed("http://example.com/rss") # Regular
<Feed: http://example.com/rss>
>>> refresh_feed.delay("http://example.com/rss") # Async
<AsyncResult: 8998d0f4-da0b-4669-ba03-d5ab5ac6ad5d>
```

---

create_task_cls()
Creates a base task class using default configuration taken from this app.

TaskSet(**args*, ***kwargs*)
Create new `TaskSet`.

send_task(*name*, *args=None*, *kwargs=None*, *countdown=None*, *eta=None*, *task_id=None*, *publisher=None*, *connection=None*, *connect_timeout=None*, *result_cls=None*, *expires=None*, *queues=None*, ***options*)
Send task by name.

Parameters

- **name** – Name of task to execute (e.g. *"tasks.add"*).
- **result_cls** – Specify custom result class. Default is using `AsyncResult()`.

Supports the same arguments as `apply_async()`.

AsyncResult(*task_id*, *backend=None*, *task_name=None*)
Create `celery.result.BaseAsyncResult` instance.

TaskSetResult(*taskset_id*, *results*, ***kwargs*)
Create `celery.result.TaskSetResult` instance.

worker_main(*argv=None*)
Run **celeryd** using *argv*. Uses `sys.argv` if *argv* is not specified.

Worker(***kwargs*)
Create new `Worker` instance.

Beat(***kwargs*)
Create new `Beat` instance.

broker_connection(*hostname=None*, *userid=None*, *password=None*, *virtual_host=None*, *port=None*, *ssl=None*, *insist=None*, *connect_timeout=None*, *transport=None*, *transport_options=None*, ***kwargs*)
Establish a connection to the message broker.

Parameters

- **hostname** – defaults to the `BROKER_HOST` setting.
- **userid** – defaults to the `BROKER_USER` setting.
- **password** – defaults to the `BROKER_PASSWORD` setting.
- **virtual_host** – defaults to the `BROKER_VHOST` setting.
- **port** – defaults to the `BROKER_PORT` setting.
- **ssl** – defaults to the `BROKER_USE_SSL` setting.
- **insist** – defaults to the `BROKER_INSIST` setting.
- **connect_timeout** – defaults to the `BROKER_CONNECTION_TIMEOUT` setting.
- **backend_cls** – defaults to the `BROKER_TRANSPORT` setting.

:returns `kombu.connection.BrokerConnection`:

with_default_connection(*fun*)
With any function accepting *connection* and *connect_timeout* keyword arguments, establishes a default connection if one is not already passed to it.

Any automatically established connection will be closed after the function returns.

Deprecated

Use `with app.default_connection(connection)` instead.

mail_admins(*subject*, *body*, *fail_silently=False*)
Send an email to the admins in the `ADMINS` setting.

prepare_config(*c*)
Prepare configuration before it is merged with the defaults.

either(*default_key*, **values*)
Fallback to the value of a configuration key if none of the **values* are true.

merge(*l*, *r*)
Like *dict(a, **b)* except it will keep values from *a* if the value in *b* is `None`.

10.1.2 Functions

`celery.app.`**app_or_default**(*app=None*)
Returns the app provided or the default app if none.

The environment variable `CELERY_TRACE_APP` is used to trace app leaks. When enabled an exception is raised if there is no active app.

10.2 celery.app.task

- celery.app.task

10.2.1 celery.app.task

Tasks Implementation.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.app.task.`**TaskType**
Meta class for tasks.

Automatically registers the task in the task registry, except if the *abstract* attribute is set.

If no *name* attribute is provided, then no name is automatically set to the name of the module it was defined in, and the class name.

class `celery.app.task.`**BaseTask**
Task base class.

When called tasks apply the `run()` method. This method must be defined by all tasks (that is unless the `__call__()` method is overridden).

classmethod AsyncResult(*task_id*)
Get AsyncResult instance for this kind of task.

Parameters **task_id** – Task id to get result for.

class ErrorMail(*task*, ***kwargs*)
Defines how and when task error e-mails should be sent.

Parameters **task** – The task instance that raised the error.

`subject` and `body` are format strings which are passed a context containing the following keys:

•name

Name of the task.

•id

UUID of the task.

•exc

String representation of the exception.

•args

Positional arguments.

•kwargs

Keyword arguments.

•traceback

String representation of the traceback.

•hostname

Worker hostname.

should_send(*context*, *exc*)
Returns true or false depending on if a task error mail should be sent for this type of error.

exception BaseTask.**MaxRetriesExceededError**
The tasks max restart limit has been exceeded.

BaseTask.**abstract = None**
If `True` the task is an abstract base class.

BaseTask.**accept_magic_kwargs = False**
If disabled the worker will not forward magic keyword arguments. Deprecated and scheduled for removal in v3.0.

BaseTask.**acks_late = False**
When enabled messages for this task will be acknowledged **after** the task has been executed, and not *just before* which is the default behavior.

Please note that this means the task may be executed twice if the worker crashes mid execution (which may be acceptable for some applications).

The application default can be overridden with the `CELERY_ACKS_LATE` setting.

BaseTask.**after_return**(*status*, *retval*, *task_id*, *args*, *kwargs*, *einfo*)
Handler called after the task returns.

Parameters

- **status** – Current task state.

- **retval** – Task return value/exception.
- **task_id** – Unique id of the task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – `ExceptionInfo` instance, containing the traceback (if any).

The return value of this handler is ignored.

`BaseTask.`**app = None**
The application instance associated with this task class.

classmethod `BaseTask.`**apply**(*args=None*, *kwargs=None*, ***options*)
Execute this task locally, by blocking until the task returns.

Parameters

- **args** – positional arguments passed on to the task.
- **kwargs** – keyword arguments passed on to the task.
- **throw** – Re-raise task exceptions. Defaults to the `CELERY_EAGER_PROPAGATES_EXCEPTIONS` setting.

:rtype `celery.result.EagerResult`:

classmethod `BaseTask.`**apply_async**(*args=None*, *kwargs=None*, *countdown=None*, *eta=None*, *task_id=None*, *publisher=None*, *connection=None*, *connect_timeout=None*, *router=None*, *expires=None*, *queues=None*, ***options*)
Apply tasks asynchronously by sending a message.

Parameters

- **args** – The positional arguments to pass on to the task (a `list` or `tuple`).
- **kwargs** – The keyword arguments to pass on to the task (a `dict`)
- **countdown** – Number of seconds into the future that the task should execute. Defaults to immediate execution (do not confuse with the *immediate* flag, as they are unrelated).
- **eta** – A `datetime` object describing the absolute time and date of when the task should be executed. May not be specified if *countdown* is also supplied. (Do not confuse this with the *immediate* flag, as they are unrelated).
- **expires** – Either a `int`, describing the number of seconds, or a `datetime` object that describes the absolute time and date of when the task should expire. The task will not be executed after the expiration time.
- **connection** – Re-use existing broker connection instead of establishing a new one. The *connect_timeout* argument is not respected if this is set.
- **connect_timeout** – The timeout in seconds, before we give up on establishing a connection to the AMQP server.
- **retry** – If enabled sending of the task message will be retried in the event of connection loss or failure. Default is taken from the `CELERY_TASK_PUBLISH_RETRY` setting. Note you need to handle the publisher/connection manually for this to work.
- **retry_policy** – Override the retry policy used. See the `CELERY_TASK_PUBLISH_RETRY` setting.

- **routing_key** – The routing key used to route the task to a worker server. Defaults to the `routing_key` attribute.
- **exchange** – The named exchange to send the task to. Defaults to the `exchange` attribute.
- **exchange_type** – The exchange type to initialize the exchange if not already declared. Defaults to the `exchange_type` attribute.
- **immediate** – Request immediate delivery. Will raise an exception if the task cannot be routed to a worker immediately. (Do not confuse this parameter with the *countdown* and *eta* settings, as they are unrelated). Defaults to the `immediate` attribute.
- **mandatory** – Mandatory routing. Raises an exception if there's no running workers able to take on this task. Defaults to the `mandatory` attribute.
- **priority** – The task priority, a number between 0 and 9. Defaults to the `priority` attribute.
- **serializer** – A string identifying the default serialization method to use. Can be *pickle*, *json*, *yaml*, *msgpack* or any custom serialization method that has been registered with `kombu.serialization.registry`. Defaults to the `serializer` attribute.
- **compression** – A string identifying the compression method to use. Can be one of `zlib`, `bzip2`, or any custom compression methods registered with `kombu.compression.register()`. Defaults to the `CELERY_MESSAGE_COMPRESSION` setting.

Note: If the `CELERY_ALWAYS_EAGER` setting is set, it will be replaced by a local `apply()` call instead.

`BaseTask.`**`autoregister = True`**
If disabled this task won't be registered automatically.

`BaseTask.`**`backend = None`**
The result store backend used for this task.

`BaseTask.`**`default_retry_delay = 180`**
Default time in seconds before a retry of the task should be executed. 3 minutes by default.

classmethod `BaseTask.`**`delay`**(**args*, ***kwargs*)
Star argument version of `apply_async()`.

Does not support the extra options enabled by `apply_async()`.

Parameters

- ***args** – positional arguments passed on to the task.
- ****kwargs** – keyword arguments passed on to the task.

:returns `celery.result.AsyncResult`:

`BaseTask.`**`delivery_mode = None`**
Override the apps default delivery mode for this task. Default is *"persistent"*, but you can change this to *"transient"*, which means messages will be lost if the broker is restarted. Consult your broker manual for any additional delivery modes.

`BaseTask.`**`error_whitelist = ()`**
List of exception types to send error emails for.

classmethod `BaseTask.`**`establish_connection`**(*connect_timeout=None*)
Establish a connection to the message broker.

BaseTask.**exchange** = None
Overrides the apps default *exchange* for this task.

BaseTask.**exchange_type** = None
Overrides the apps default exchange type for this task.

BaseTask.**execute**(*request*, *pool*, *loglevel*, *logfile*, ***kwargs*)
The method the worker calls to execute the task.

Parameters

- **request** – A TaskRequest.
- **pool** – A task pool.
- **loglevel** – Current loglevel.
- **logfile** – Name of the currently used logfile.
- **consumer** – The Consumer.

BaseTask.**expires** = None
Default task expiry time.

classmethod BaseTask.**get_consumer**(*connection=None*, *connect_timeout=None*)
Get message consumer.

:rtype kombu.messaging.Consumer:

Warning: If you don't specify a connection, one will automatically be established for you, in that case you need to close this connection after use:

```
>>> consumer = self.get_consumer()
>>> # do something with consumer
>>> consumer.close()
>>> consumer.connection.close()
```

classmethod BaseTask.**get_logger**(*loglevel=None*, *logfile=None*, *propagate=False*, ***kwargs*)
Get task-aware logger object.

classmethod BaseTask.**get_publisher**(*connection=None*, *exchange=None*, *connect_timeout=None*, *exchange_type=None*, ***options*)
Get a celery task message publisher.

:rtype TaskPublisher:

Warning: If you don't specify a connection, one will automatically be established for you, in that case you need to close this connection after use:

```
>>> publisher = self.get_publisher()
>>> # ... do something with publisher
>>> publisher.connection.close()
```

or used as a context:

```
>>> with self.get_publisher() as publisher:
...     # ... do something with publisher
```

BaseTask.**ignore_result** = False
If enabled the worker will not store task state and return values for this task. Defaults to the CELERY_IGNORE_RESULT setting.

BaseTask.**immediate** = False
Request immediate delivery.

BaseTask.**mandatory** = False
Mandatory message routing.

BaseTask.**max_retries** = 3
Maximum number of retries before giving up. If set to None, it will **never** stop retrying.

BaseTask.**name** = None
Name of the task.

BaseTask.**on_failure**(*exc*, *task_id*, *args*, *kwargs*, *einfo*)
Error handler.

This is run by the worker when the task fails.

Parameters

- **exc** – The exception raised by the task.
- **task_id** – Unique id of the failed task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – ExceptionInfo instance, containing the traceback.

The return value of this handler is ignored.

BaseTask.**on_retry**(*exc*, *task_id*, *args*, *kwargs*, *einfo*)
Retry handler.

This is run by the worker when the task is to be retried.

Parameters

- **exc** – The exception sent to retry().
- **task_id** – Unique id of the retried task.
- **args** – Original arguments for the retried task.
- **kwargs** – Original keyword arguments for the retried task.
- **einfo** – ExceptionInfo instance, containing the traceback.

The return value of this handler is ignored.

BaseTask.**on_success**(*retval*, *task_id*, *args*, *kwargs*)
Success handler.

Run by the worker if the task executes successfully.

Parameters

- **retval** – The return value of the task.
- **task_id** – Unique id of the executed task.
- **args** – Original arguments for the executed task.
- **kwargs** – Original keyword arguments for the executed task.

The return value of this handler is ignored.

`BaseTask.`**`priority`** **= None**
Default message priority. A number between 0 to 9, where 0 is the highest. Note that RabbitMQ does not support priorities.

`BaseTask.`**`queue`** **= None**
Destination queue. The queue needs to exist in `CELERY_QUEUES`. The *routing_key*, *exchange* and *exchange_type* attributes will be ignored if this is set.

`BaseTask.`**`rate_limit`** **= None**
Rate limit for this task type. Examples: `None` (no rate limit), *"100/s"* (hundred tasks a second), *"100/m"* (hundred tasks a minute),‘"100/h"‘ (hundred tasks an hour)

`BaseTask.`**`request`** **= <celery.app.task.Context object at 0x7f6b808>**
Request context (set when task is applied).

classmethod `BaseTask.`**`retry`**(*args=None*, *kwargs=None*, *exc=None*, *throw=True*, *eta=None*, *countdown=None*, *max_retries=None*, ***options*)
Retry the task.

Parameters

- **args** – Positional arguments to retry with.
- **kwargs** – Keyword arguments to retry with.
- **exc** – Optional exception to raise instead of `MaxRetriesExceededError` when the max restart limit has been exceeded.
- **countdown** – Time in seconds to delay the retry for.
- **eta** – Explicit time and date to run the retry at (must be a `datetime` instance).
- **max_retries** – If set, overrides the default retry limit.
- ****options** – Any extra options to pass on to meth:*apply_async*.
- **throw** – If this is `False`, do not raise the `RetryTaskError` exception, that tells the worker to mark the task as being retried. Note that this means the task will be marked as failed if the task raises an exception, or successful if it returns.

Raises celery.exceptions.RetryTaskError To tell the worker that the task has been re-sent for retry. This always happens, unless the *throw* keyword argument has been explicitly set to `False`, and is considered normal operation.

Example

```
>>> @task
>>> def tweet(auth, message):
...     twitter = Twitter(oauth=auth)
...     try:
...         twitter.post_status_update(message)
...     except twitter.FailWhale, exc:
...         # Retry in 5 minutes.
...         return tweet.retry(countdown=60 * 5, exc=exc)
```

Although the task will never return above as *retry* raises an exception to notify the worker, we use *return* in front of the retry to convey that the rest of the block will not be executed.

`BaseTask.`**`routing_key`** **= None**
Overrides the apps default *routing_key* for this task.

BaseTask.**run**(**args*, ***kwargs*)
The body of the task executed by workers.

BaseTask.**send_error_emails = False**
If enabled an email will be sent to ADMINS whenever a task of this type fails.

BaseTask.**serializer = 'pickle'**
The name of a serializer that are registered with kombu.serialization.registry. Default is *"pickle"*.

BaseTask.**soft_time_limit = None**
Soft time limit. Defaults to the CELERY_TASK_SOFT_TIME_LIMIT setting.

BaseTask.**store_errors_even_if_ignored = False**
When enabled errors will be stored even if the task is otherwise configured to ignore results.

classmethod BaseTask.**subtask**(**args*, ***kwargs*)
Returns subtask object for this task, wrapping arguments and execution options for a single task invocation.

BaseTask.**time_limit = None**
Hard time limit. Defaults to the CELERY_TASK_TIME_LIMIT setting.

BaseTask.**track_started = False**
If enabled the task will report its status as "started" when the task is executed by a worker. Disabled by default as the normal behaviour is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried.

Having a "started" status can be useful for when there are long running tasks and there is a need to report which task is currently running.

The application default can be overridden using the CELERY_TRACK_STARTED setting.

BaseTask.**type = 'regular'**
The type of task *(no longer used)*.

BaseTask.**update_state**(*task_id=None*, *state=None*, *meta=None*)
Update task state.

Parameters

- **task_id** – Id of the task to update.
- **state** – New state (str).
- **meta** – State metadata (dict).

10.3 celery.app.amqp

AMQ related functionality.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

- AMQP
- Queues
- TaskPublisher

10.3.1 AMQP

class celery.app.amqp.**AMQP**(*app*)

BrokerConnection
Broker connection class used. Default is kombu.connection.BrokerConnection.

Consumer
The task consumer class used. Default is kombu.compat.Consumer.

ConsumerSet
The class used to consume from multiple queues using the same channel.

queues
All currently defined task queues. (A Queues instance).

Queues(*queues*)
Create new Queues instance, using queue defaults from the current configuration.

Router(*queues=None*, *create_missing=None*)
Returns the current task router.

TaskConsumer(**args*, ***kwargs*)
Returns consumer for a single task queue.

TaskPublisher(**args*, ***kwargs*)
Returns publisher used to send tasks.

You should use *app.send_task* instead.

get_task_consumer(*connection*, *queues=None*, ***kwargs*)
Return consumer configured to consume from all known task queues.

get_default_queue()
Returns *(queue_name, queue_options)* tuple for the queue configured to be default (CELERY_DEFAULT_QUEUE).

10.3.2 Queues

class celery.app.amqp.**Queues**(*queues*)
Queue name declaration mapping.

Celery will consult this mapping to find the options for any queue by name.

Parameters **queues** – Initial mapping.

add(*queue*, *exchange=None*, *routing_key=None*, *exchange_type='direct'*, ***options*)
Add new queue.

Parameters

- **queue** – Name of the queue.
- **exchange** – Name of the exchange.
- **routing_key** – Binding key.
- **exchange_type** – Type of exchange.
- ****options** – Additional declaration options.

options(*exchange*, *routing_key*, *exchange_type='direct'*, ***options*)
Creates new option mapping for queue, with required keys present.

format(*indent=0*, *indent_first=True*)
Format routing table into string for log dumps.

select_subset(*wanted*, *create_missing=True*)
Select subset of the currently defined queues.

Does not return anything: queues not in *wanted* will be discarded in-place.

Parameters

- **wanted** – List of wanted queue names.
- **create_missing** – By default any unknown queues will be added automatically, but if disabled the occurrence of unknown queues in *wanted* will raise `KeyError`.

classmethod with_defaults(*queues*, *default_exchange*, *default_exchange_type*)
Alternate constructor that adds default exchange and exchange type information to queues that does not have any.

10.3.3 TaskPublisher

class celery.app.amqp.**TaskPublisher**(**args*, ***kwargs*)

auto_declare = True

declare()

delay_task(*task_name*, *task_args=None*, *task_kwargs=None*, *countdown=None*, *eta=None*, *task_id=None*, *taskset_id=None*, *expires=None*, *exchange=None*, *exchange_type=None*, *event_dispatcher=None*, *retry=None*, *retry_policy=None*, *queue=None*, *now=None*, *retries=0*, *chord=None*, ***kwargs*)
Send task message.

retry = False

retry_policy = None

10.4 celery.app.defaults

- celery.app.defaults

10.4.1 celery.app.defaults

Configuration introspection and defaults.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.app.defaults.**Option**(*default=None*, **args*, ***kwargs*)

alt = None

deprecate_by = None

remove_by = None

to_python(*value*)

typemap = {'bool': <function str_to_bool at 0x2f128c0>, 'string': <type 'str'>, 'tuple': <type 'tuple'>, 'int': <type 'int'>,

celery.app.defaults.**find_deprecated_settings**(*source*)

celery.app.defaults.**flatten**(*d*, *ns=''*)

celery.app.defaults.**str_to_bool**(*term*, *table={'false': False, 'no': False, '1': True, '0': False, 'yes': True, 'true': True}*)

10.5 celery.task

- celery.task

10.5.1 celery.task

Creating tasks, subtasks, sets and chords.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

celery.task.**task**(**args*, ***kwargs*)

Decorator to create a task class out of any callable.

Examples

```
@task
def refresh_feed(url):
    return Feed.objects.get(url=url).refresh()
```

With setting extra options and using retry.

```
@task(max_retries=10)
def refresh_feed(url):
    try:
        return Feed.objects.get(url=url).refresh()
    except socket.error, exc:
        refresh_feed.retry(exc=exc)
```

Calling the resulting task:

```
>>> refresh_feed("http://example.com/rss") # Regular
<Feed: http://example.com/rss>
>>> refresh_feed.delay("http://example.com/rss") # Async
<AsyncResult: 8998d0f4-da0b-4669-ba03-d5ab5ac6ad5d>
```

celery.task.**periodic_task**(**args*, ***options*)
Decorator to create a task class out of any callable.

Examples

```
@task
def refresh_feed(url):
    return Feed.objects.get(url=url).refresh()
```

With setting extra options and using retry.

```
@task(exchange="feeds")
def refresh_feed(url, **kwargs):
    try:
        return Feed.objects.get(url=url).refresh()
    except socket.error, exc:
        refresh_feed.retry(args=[url], kwargs=kwargs, exc=exc)
```

Calling the resulting task:

```
>>> refresh_feed("http://example.com/rss") # Regular
<Feed: http://example.com/rss>
>>> refresh_feed.delay("http://example.com/rss") # Async
<AsyncResult: 8998d0f4-da0b-4669-ba03-d5ab5ac6ad5d>
```

---

class celery.task.**Task**
Task base class.

When called tasks apply the `run()` method. This method must be defined by all tasks (that is unless the `__call__()` method is overridden).

See also:

`celery.task.base.BaseTask`.

10.6 celery.task.base

- celery.task.base

10.6.1 celery.task.base

The task implementation has been moved to `celery.app.task`.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.task.base.**BaseTask**
Task base class.

When called tasks apply the `run()` method. This method must be defined by all tasks (that is unless the `__call__()` method is overridden).

classmethod **AsyncResult**(*task_id*)
Get AsyncResult instance for this kind of task.

Parameters **task_id** – Task id to get result for.

class **ErrorMail**(*task*, ***kwargs*)
Defines how and when task error e-mails should be sent.

Parameters **task** – The task instance that raised the error.

`subject` and `body` are format strings which are passed a context containing the following keys:

•name

Name of the task.

•id

UUID of the task.

•exc

String representation of the exception.

•args

Positional arguments.

•kwargs

Keyword arguments.

•traceback

String representation of the traceback.

•hostname

Worker hostname.

should_send(*context*, *exc*)
Returns true or false depending on if a task error mail should be sent for this type of error.

exception BaseTask.**MaxRetriesExceededError**
The tasks max restart limit has been exceeded.

BaseTask.**after_return**(*status*, *retval*, *task_id*, *args*, *kwargs*, *einfo*)
Handler called after the task returns.

Parameters

- **status** – Current task state.
- **retval** – Task return value/exception.
- **task_id** – Unique id of the task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – `ExceptionInfo` instance, containing the traceback (if any).

The return value of this handler is ignored.

classmethod BaseTask.**apply**(*args=None*, *kwargs=None*, ***options*)
Execute this task locally, by blocking until the task returns.

Parameters

- **args** – positional arguments passed on to the task.
- **kwargs** – keyword arguments passed on to the task.
- **throw** – Re-raise task exceptions. Defaults to the `CELERY_EAGER_PROPAGATES_EXCEPTIONS` setting.

:rtype `celery.result.EagerResult`:

classmethod `BaseTask.`**`apply_async`**(*args=None*, *kwargs=None*, *countdown=None*, *eta=None*, *task_id=None*, *publisher=None*, *connection=None*, *connect_timeout=None*, *router=None*, *expires=None*, *queues=None*, ***options*)

Apply tasks asynchronously by sending a message.

Parameters

- **args** – The positional arguments to pass on to the task (a `list` or `tuple`).
- **kwargs** – The keyword arguments to pass on to the task (a `dict`)
- **countdown** – Number of seconds into the future that the task should execute. Defaults to immediate execution (do not confuse with the *immediate* flag, as they are unrelated).
- **eta** – A `datetime` object describing the absolute time and date of when the task should be executed. May not be specified if *countdown* is also supplied. (Do not confuse this with the *immediate* flag, as they are unrelated).
- **expires** – Either a `int`, describing the number of seconds, or a `datetime` object that describes the absolute time and date of when the task should expire. The task will not be executed after the expiration time.
- **connection** – Re-use existing broker connection instead of establishing a new one. The *connect_timeout* argument is not respected if this is set.
- **connect_timeout** – The timeout in seconds, before we give up on establishing a connection to the AMQP server.
- **retry** – If enabled sending of the task message will be retried in the event of connection loss or failure. Default is taken from the `CELERY_TASK_PUBLISH_RETRY` setting. Note you need to handle the publisher/connection manually for this to work.
- **retry_policy** – Override the retry policy used. See the `CELERY_TASK_PUBLISH_RETRY` setting.
- **routing_key** – The routing key used to route the task to a worker server. Defaults to the `routing_key` attribute.
- **exchange** – The named exchange to send the task to. Defaults to the `exchange` attribute.
- **exchange_type** – The exchange type to initialize the exchange if not already declared. Defaults to the `exchange_type` attribute.
- **immediate** – Request immediate delivery. Will raise an exception if the task cannot be routed to a worker immediately. (Do not confuse this parameter with the *countdown* and *eta* settings, as they are unrelated). Defaults to the `immediate` attribute.
- **mandatory** – Mandatory routing. Raises an exception if there's no running workers able to take on this task. Defaults to the `mandatory` attribute.
- **priority** – The task priority, a number between 0 and 9. Defaults to the `priority` attribute.

- **serializer** – A string identifying the default serialization method to use. Can be *pickle*, *json*, *yaml*, *msgpack* or any custom serialization method that has been registered with `kombu.serialization.registry`. Defaults to the `serializer` attribute.
- **compression** – A string identifying the compression method to use. Can be one of `zlib`, `bzip2`, or any custom compression methods registered with `kombu.compression.register()`. Defaults to the `CELERY_MESSAGE_COMPRESSION` setting.

Note: If the `CELERY_ALWAYS_EAGER` setting is set, it will be replaced by a local `apply()` call instead.

---

classmethod `BaseTask.`**`delay`**(**args*, ***kwargs*)
Star argument version of `apply_async()`.

Does not support the extra options enabled by `apply_async()`.

Parameters

- ***args** – positional arguments passed on to the task.
- ****kwargs** – keyword arguments passed on to the task.

:returns `celery.result.AsyncResult`:

classmethod `BaseTask.`**`establish_connection`**(*connect_timeout=None*)
Establish a connection to the message broker.

`BaseTask.`**`execute`**(*request*, *pool*, *loglevel*, *logfile*, ***kwargs*)
The method the worker calls to execute the task.

Parameters

- **request** – A `TaskRequest`.
- **pool** – A task pool.
- **loglevel** – Current loglevel.
- **logfile** – Name of the currently used logfile.
- **consumer** – The `Consumer`.

classmethod `BaseTask.`**`get_consumer`**(*connection=None*, *connect_timeout=None*)
Get message consumer.

:rtype `kombu.messaging.Consumer`:

Warning: If you don't specify a connection, one will automatically be established for you, in that case you need to close this connection after use:

```
>>> consumer = self.get_consumer()
>>> # do something with consumer
>>> consumer.close()
>>> consumer.connection.close()
```

classmethod `BaseTask.`**`get_logger`**(*loglevel=None*, *logfile=None*, *propagate=False*, ***kwargs*)
Get task-aware logger object.

classmethod `BaseTask.`**`get_publisher`**(*connection=None*, *exchange=None*, *connect_timeout=None*, *exchange_type=None*, ***options*)
Get a celery task message publisher.

:rtype TaskPublisher:

Warning: If you don't specify a connection, one will automatically be established for you, in that case you need to close this connection after use:

```
>>> publisher = self.get_publisher()
>>> # ... do something with publisher
>>> publisher.connection.close()
```

or used as a context:

```
>>> with self.get_publisher() as publisher:
...     # ... do something with publisher
```

BaseTask.**on_failure**(*exc*, *task_id*, *args*, *kwargs*, *einfo*)
Error handler.

This is run by the worker when the task fails.

Parameters

- **exc** – The exception raised by the task.
- **task_id** – Unique id of the failed task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – ExceptionInfo instance, containing the traceback.

The return value of this handler is ignored.

BaseTask.**on_retry**(*exc*, *task_id*, *args*, *kwargs*, *einfo*)
Retry handler.

This is run by the worker when the task is to be retried.

Parameters

- **exc** – The exception sent to retry().
- **task_id** – Unique id of the retried task.
- **args** – Original arguments for the retried task.
- **kwargs** – Original keyword arguments for the retried task.
- **einfo** – ExceptionInfo instance, containing the traceback.

The return value of this handler is ignored.

BaseTask.**on_success**(*retval*, *task_id*, *args*, *kwargs*)
Success handler.

Run by the worker if the task executes successfully.

Parameters

- **retval** – The return value of the task.
- **task_id** – Unique id of the executed task.
- **args** – Original arguments for the executed task.
- **kwargs** – Original keyword arguments for the executed task.

The return value of this handler is ignored.

classmethod BaseTask.**retry**(*args=None*, *kwargs=None*, *exc=None*, *throw=True*, *eta=None*, *countdown=None*, *max_retries=None*, ***options*)

Retry the task.

Parameters

- **args** – Positional arguments to retry with.
- **kwargs** – Keyword arguments to retry with.
- **exc** – Optional exception to raise instead of MaxRetriesExceededError when the max restart limit has been exceeded.
- **countdown** – Time in seconds to delay the retry for.
- **eta** – Explicit time and date to run the retry at (must be a datetime instance).
- **max_retries** – If set, overrides the default retry limit.
- ****options** – Any extra options to pass on to meth:*apply_async*.
- **throw** – If this is False, do not raise the RetryTaskError exception, that tells the worker to mark the task as being retried. Note that this means the task will be marked as failed if the task raises an exception, or successful if it returns.

Raises celery.exceptions.RetryTaskError To tell the worker that the task has been re-sent for retry. This always happens, unless the *throw* keyword argument has been explicitly set to False, and is considered normal operation.

Example

```
>>> @task
>>> def tweet(auth, message):
...     twitter = Twitter(oauth=auth)
...     try:
...         twitter.post_status_update(message)
...     except twitter.FailWhale, exc:
...         # Retry in 5 minutes.
...         return tweet.retry(countdown=60 * 5, exc=exc)
```

Although the task will never return above as *retry* raises an exception to notify the worker, we use *return* in front of the retry to convey that the rest of the block will not be executed.

BaseTask.**run**(**args*, ***kwargs*)

The body of the task executed by workers.

classmethod BaseTask.**subtask**(**args*, ***kwargs*)

Returns subtask object for this task, wrapping arguments and execution options for a single task invocation.

BaseTask.**update_state**(*task_id=None*, *state=None*, *meta=None*)

Update task state.

Parameters

- **task_id** – Id of the task to update.
- **state** – New state (str).
- **meta** – State metadata (dict).

class celery.task.base.**PeriodicTask**

A periodic task is a task that behaves like a *cron* job.

Results of periodic tasks are not stored by default.

run_every
REQUIRED Defines how often the task is run (its interval), it can be a `timedelta` object, a `crontab` object or an integer specifying the time in seconds.

relative
If set to `True`, run times are relative to the time when the server was started. This was the previous behaviour, periodic tasks are now scheduled by the clock.

Raises **NotImplementedError** if the `run_every` attribute is not defined.

Example

```
>>> from celery.task import tasks, PeriodicTask
>>> from datetime import timedelta
>>> class EveryThirtySecondsTask(PeriodicTask):
...     run_every = timedelta(seconds=30)
...
...     def run(self, **kwargs):
...         logger = self.get_logger(**kwargs)
...         logger.info("Execute every 30 seconds")

>>> from celery.task import PeriodicTask
>>> from celery.schedules import crontab

>>> class EveryMondayMorningTask(PeriodicTask):
...     run_every = crontab(hour=7, minute=30, day_of_week=1)
...
...     def run(self, **kwargs):
...         logger = self.get_logger(**kwargs)
...         logger.info("Execute every Monday at 7:30AM.")

>>> class EveryMorningTask(PeriodicTask):
...     run_every = crontab(hours=7, minute=30)
...
...     def run(self, **kwargs):
...         logger = self.get_logger(**kwargs)
...         logger.info("Execute every day at 7:30AM.")

>>> class EveryQuarterPastTheHourTask(PeriodicTask):
...     run_every = crontab(minute=15)
...
...     def run(self, **kwargs):
...         logger = self.get_logger(**kwargs)
...         logger.info("Execute every 0:15 past the hour every day.")
```

is_due(*last_run_at*)
Returns tuple of two items *(is_due, next_time_to_run)*, where next time to run is in seconds.

See `celery.schedules.schedule.is_due()` for more information.

remaining_estimate(*last_run_at*)
Returns when the periodic task should run next as a timedelta.

timedelta_seconds(*delta*)
Convert `timedelta` to seconds.

Doesn't account for negative timedeltas.

class celery.task.base.**TaskType**
Meta class for tasks.

Automatically registers the task in the task registry, except if the *abstract* attribute is set.

If no *name* attribute is provided, then no name is automatically set to the name of the module it was defined in, and the class name.

10.7 celery.task.sets

- celery.task.sets

10.7.1 celery.task.sets

Creating and applying groups of tasks.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.task.sets.**TaskSet**(*task=None*, *tasks=None*, *app=None*, *Publisher=None*)
A task containing several subtasks, making it possible to track how many, or when all of the tasks have been completed.

Parameters **tasks** – A list of subtask instances.

Example:

```
>>> urls = ("http://cnn.com/rss", "http://bbc.co.uk/rss")
>>> taskset = TaskSet(refresh_feed.subtask((url, )) for url in urls)
>>> taskset_result = taskset.apply_async()
>>> list_of_return_values = taskset_result.join()  # *expensive*
```

apply(*taskset_id=None*)
Applies the taskset locally by blocking until all tasks return.

apply_async(*connection=None*, *connect_timeout=None*, *publisher=None*, *taskset_id=None*)
Apply taskset.

task

task_name

tasks

total = None
Total number of subtasks in this set.

celery.task.sets.**maybe_subtask**(*t*)

class celery.task.sets.**subtask**(*task=None*, *args=None*, *kwargs=None*, *options=None*, ***ex*)
Class that wraps the arguments and execution options for a single task invocation.

Used as the parts in a TaskSet or to safely pass tasks around as callbacks.

Parameters

- **task** – Either a task class/instance, or the name of a task.
- **args** – Positional arguments to apply.
- **kwargs** – Keyword arguments to apply.
- **options** – Additional options to `celery.execute.apply_async()`.

Note that if the first argument is a `dict`, the other arguments will be ignored and the values in the dict will be used instead.

```
>>> s = subtask("tasks.add", args=(2, 2))
>>> subtask(s)
{"task": "tasks.add", args=(2, 2), kwargs={}, options={}}
```

apply(*args=()*, *kwargs={}*, ***options*)
Apply this task locally.

apply_async(*args=()*, *kwargs={}*, ***options*)
Apply this task asynchronously.

delay(**argmerge*, ***kwmerge*)
Shortcut to *apply_async(argmerge, kwargs)*.

type

10.8 celery.task.chords

- celery.task.chords

10.8.1 celery.task.chords

Chords (task set callbacks).

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.task.chords.`**Chord**

accept_magic_kwargs = False

name = 'celery.chord'

run(*set*, *body*, *interval=1*, *max_retries=None*, *propagate=False*, ***kwargs*)

class `celery.task.chords.`**chord**(*tasks*, ***options*)

class Chord

accept_magic_kwargs = False

name = 'celery.chord'

run(*set*, *body*, *interval=1*, *max_retries=None*, *propagate=False*, ***kwargs*)

10.9 celery.result

- celery.result

10.9.1 celery.result

Task results/state and groups of results.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.result.`**`AsyncResult`**(*task_id*, *backend=None*, *task_name=None*, *app=None*)
Pending task result using the default backend.

Parameters **task_id** – The task uuid.

`backend` **= None**
Task result store backend to use.

class `celery.result.`**`BaseAsyncResult`**(*task_id*, *backend*, *task_name=None*, *app=None*)
Base class for pending result, supports custom task result backend.

Parameters

- **task_id** – see `task_id`.
- **backend** – see `backend`.

exception `TimeoutError`
Error raised for timeouts.

`BaseAsyncResult.`**`backend`** **= None**
The task result backend to use.

`BaseAsyncResult.`**`failed`**`()`
Returns `True` if the task failed.

`BaseAsyncResult.`**`forget`**`()`
Forget about (and possibly remove the result of) this task.

`BaseAsyncResult.`**`get`**(*timeout=None*, *propagate=True*, *interval=0.5*)
Wait until task is ready, and return its result.

> **Warning:** Waiting for tasks within a task may lead to deadlocks. Please read *Avoid launching synchronous subtasks*.

Parameters

- **timeout** – How long to wait, in seconds, before the operation times out.
- **propagate** – Re-raise exception if the task failed.
- **interval** – Time to wait (in seconds) before retrying to retrieve the result. Note that this does not have any effect when using the AMQP result store backend, as it does not use polling.

Raises celery.exceptions.TimeoutError if *timeout* is not `None` and the result does not arrive within *timeout* seconds.

If the remote call raised an exception then that exception will be re-raised.

`BaseAsyncResult.`**`info`**
Get state metadata. Alias to `result()`.

`BaseAsyncResult.`**`ready`**`()`
Returns `True` if the task has been executed.

If the task is still running, pending, or is waiting for retry then `False` is returned.

`BaseAsyncResult.`**`result`**
When the task has been executed, this contains the return value. If the task raised an exception, this will be the exception instance.

`BaseAsyncResult.`**`revoke`**`(`*connection=None*, *connect_timeout=None*`)`
Send revoke signal to all workers.

Any worker receiving the task, or having reserved the task, *must* ignore it.

`BaseAsyncResult.`**`state`**
The tasks current state.

Possible values includes:

PENDING

The task is waiting for execution.

STARTED

The task has been started.

RETRY

The task is to be retried, possibly because of failure.

FAILURE

The task raised an exception, or has exceeded the retry limit. The `result` attribute then contains the exception raised by the task.

SUCCESS

The task executed successfully. The `result` attribute then contains the tasks return value.

`BaseAsyncResult.`**`status`**
Deprecated alias of `state`.

`BaseAsyncResult.`**`successful`**`()`
Returns `True` if the task executed successfully.

`BaseAsyncResult.`**`task_id`** **= None**
The task uuid.

`BaseAsyncResult.`**`traceback`**
Get the traceback of a failed task.

`BaseAsyncResult.`**`wait`**`(`**args*, ***kwargs*`)`
Deprecated alias to `get()`.

class `celery.result.`**`EagerResult`**`(`*task_id*, *ret_value*, *state*, *traceback=None*`)`
Result that we know has already been executed.

exception `TimeoutError`
The operation timed out.

`EagerResult.`**`get`**`(`*timeout=None*, *propagate=True*, ***kwargs*`)`
Wait until the task has been executed and return its result.

`EagerResult.`**`ready`**`()`
Returns `True` if the task has been executed.

`EagerResult.`**`result`**
The tasks return value

`EagerResult.`**`revoke`**`()`

`EagerResult.`**`state`**
The tasks state.

`EagerResult.`**`status`**
The tasks status (alias to `state`).

`EagerResult.`**`successful`**`()`
Returns `True` if the task executed without failure.

`EagerResult.`**`traceback`**
The traceback if the task failed.

class `celery.result.`**`ResultSet`**`(`*results*, *app=None*, ***kwargs*`)`
Working with more than one result.

Parameters **results** – List of result instances.

`add``(`*result*`)`
Add `AsyncResult` as a new member of the set.

Does nothing if the result is already a member.

`clear``()`
Remove all results from this set.

`completed_count``()`
Task completion count.

Returns the number of tasks completed.

`discard``(`*result*`)`
Remove result from the set if it is a member.

If it is not a member, do nothing.

`failed``()`
Did any of the tasks fail?

Returns `True` if any of the tasks failed. (i.e., raised an exception)

`forget``()`
Forget about (and possible remove the result of) all the tasks.

`iter_native``(`*timeout=None*, *interval=None*`)`
Backend optimized version of `iterate()`.

New in version 2.2.

Note that this does not support collecting the results for different task types using different backends.

This is currently only supported by the AMQP, Redis and cache result backends.

iterate(*timeout=None*, *propagate=True*, *interval=0.5*)
Iterate over the return values of the tasks as they finish one by one.

Raises The exception if any of the tasks raised an exception.

join(*timeout=None*, *propagate=True*, *interval=0.5*)
Gathers the results of all tasks as a list in order.

Note: This can be an expensive operation for result store backends that must resort to polling (e.g. database).

You should consider using `join_native()` if your backend supports it.

Warning: Waiting for tasks within a task may lead to deadlocks. Please see *Avoid launching synchronous subtasks*.

Parameters

- **timeout** – The number of seconds to wait for results before the operation times out.
- **propagate** – If any of the tasks raises an exception, the exception will be re-raised.
- **interval** – Time to wait (in seconds) before retrying to retrieve a result from the set. Note that this does not have any effect when using the AMQP result store backend, as it does not use polling.

Raises celery.exceptions.TimeoutError if *timeout* is not `None` and the operation takes longer than *timeout* seconds.

join_native(*timeout=None*, *propagate=True*, *interval=0.5*)
Backend optimized version of `join()`.

New in version 2.2.

Note that this does not support collecting the results for different task types using different backends.

This is currently only supported by the AMQP, Redis and cache result backends.

ready()
Did all of the tasks complete? (either by success of failure).

Returns `True` if all of the tasks been executed.

remove(*result*)
Removes result from the set; it must be a member.

Raises KeyError if the result is not a member.

results = None
List of results in in the set.

revoke(*connection=None*, *connect_timeout=None*)
Revoke all tasks in the set.

subtasks
Deprecated alias to `results`.

successful()
Was all of the tasks successful?

Returns `True` if all of the tasks finished successfully (i.e. did not raise an exception).

total
Total number of tasks in the set.

update(*results*)
Update set with the union of itself and an iterable with results.

waiting()
Are any of the tasks incomplete?

Returns `True` if any of the tasks is still waiting for execution.

class celery.result.**TaskSetResult**(*taskset_id*, *results=None*, ***kwargs*)
An instance of this class is returned by *TaskSet*'s `apply_async()` method.

It enables inspection of the tasks state and return values as a single entity.

Parameters

- **taskset_id** – The id of the taskset.
- **results** – List of result instances.

delete(*backend=None*)
Remove this result if it was previously saved.

itersubtasks()
Depreacted. Use `iter(self.results)` instead.

classmethod restore(*taskset_id*, *backend=None*)
Restore previously saved taskset result.

results = None
List/iterator of results in the taskset

save(*backend=None*)
Save taskset result for later retrieval using `restore()`.

Example:

```
>>> result.save()
>>> result = TaskSetResult.restore(taskset_id)
```

taskset_id = None
The UUID of the taskset.

10.10 celery.task.control

- celery.task.control

10.10.1 celery.task.control

Client for worker remote control commands. Server implementation is in `celery.worker.control`.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.task.control.**Control**(*app*)

class **Mailbox**(*namespace*, *type='direct'*, *connection=None*)

Node(*hostname=None*, *state=None*, *channel=None*, *handlers=None*)

abcast(*command*, *kwargs={}*)

call(*destination*, *command*, *kwargs={}*, *timeout=None*, *callback=None*, *channel=None*)

cast(*destination*, *command*, *kwargs={}*)

connection = None

exchange = None

exchange_fmt = '%s.pidbox'

get_queue(*hostname*)

get_reply_queue(*ticket*)

multi_call(*command*, *kwargs={}*, *timeout=1*, *limit=None*, *callback=None*, *channel=None*)

namespace = None

node_cls
alias of `Node`

reply_exchange = None

reply_exchange_fmt = 'reply.%s.pidbox'

type = 'direct'

`Control.`**broadcast**(*command*, *arguments=None*, *destination=None*, *connection=None*, *connect_timeout=None*, *reply=False*, *timeout=1*, *limit=None*, *callback=None*, *channel=None*)
Broadcast a control command to the celery workers.

Parameters

- **command** – Name of command to send.
- **arguments** – Keyword arguments for the command.
- **destination** – If set, a list of the hosts to send the command to, when empty broadcast to all workers.
- **connection** – Custom broker connection to use, if not set, a connection will be established automatically.
- **connect_timeout** – Timeout for new connection if a custom connection is not provided.
- **reply** – Wait for and return the reply.
- **timeout** – Timeout in seconds to wait for the reply.
- **limit** – Limit number of replies.
- **callback** – Callback called immediately for each reply received.

`Control.`**discard_all**(*connection=None*, *connect_timeout=None*)
Discard all waiting tasks.

This will ignore all tasks waiting for execution, and they will be deleted from the messaging server.

Returns the number of tasks discarded.

`Control.`**inspect**(*destination=None*, *timeout=1*, *callback=None*)

Control.**ping**(*destination=None*, *timeout=1*, ***kwargs*)
Ping workers.

Returns answer from alive workers.

Parameters

- **destination** – If set, a list of the hosts to send the command to, when empty broadcast to all workers.
- **connection** – Custom broker connection to use, if not set, a connection will be established automatically.
- **connect_timeout** – Timeout for new connection if a custom connection is not provided.
- **reply** – Wait for and return the reply.
- **timeout** – Timeout in seconds to wait for the reply.
- **limit** – Limit number of replies.

Control.**rate_limit**(*task_name*, *rate_limit*, *destination=None*, ***kwargs*)
Set rate limit for task by type.

Parameters

- **task_name** – Name of task to change rate limit for.
- **rate_limit** – The rate limit as tasks per second, or a rate limit string (*"100/m"*, etc. see `celery.task.base.Task.rate_limit` for more information).
- **destination** – If set, a list of the hosts to send the command to, when empty broadcast to all workers.
- **connection** – Custom broker connection to use, if not set, a connection will be established automatically.
- **connect_timeout** – Timeout for new connection if a custom connection is not provided.
- **reply** – Wait for and return the reply.
- **timeout** – Timeout in seconds to wait for the reply.
- **limit** – Limit number of replies.

Control.**revoke**(*task_id*, *destination=None*, *terminate=False*, *signal='SIGTERM'*, ***kwargs*)
Revoke a task by id.

If a task is revoked, the workers will ignore the task and not execute it after all.

Parameters

- **task_id** – Id of the task to revoke.
- **terminate** – Also terminate the process currently working on the task (if any).
- **signal** – Name of signal to send to process if terminate. Default is TERM.
- **destination** – If set, a list of the hosts to send the command to, when empty broadcast to all workers.
- **connection** – Custom broker connection to use, if not set, a connection will be established automatically.
- **connect_timeout** – Timeout for new connection if a custom connection is not provided.
- **reply** – Wait for and return the reply.

- **timeout** – Timeout in seconds to wait for the reply.
- **limit** – Limit number of replies.

`Control.`**`time_limit`**(*task_name*, *soft=None*, *hard=None*, ***kwargs*)

Set time limits for task by type.

Parameters

- **task_name** – Name of task to change time limits for.
- **soft** – New soft time limit (in seconds).
- **hard** – New hard time limit (in seconds).

Any additional keyword arguments are passed on to `broadcast()`.

class `celery.task.control.`**`Inspect`**(*control*, *destination=None*, *timeout=1*, *callback=None*)

`active`(*safe=False*)

`active_queues`()

`add_consumer`(*queue*, *exchange=None*, *exchange_type='direct'*, *routing_key=None*, ***options*)

`cancel_consumer`(*queue*, ***kwargs*)

`disable_events`()

`enable_events`()

`ping`()

`registered`()

`registered_tasks`()

`reserved`(*safe=False*)

`revoked`()

`scheduled`(*safe=False*)

`stats`()

`celery.task.control.`**`flatten_reply`**(*reply*)

10.11 celery.task.http

- celery.task.http

10.11.1 celery.task.http

Task webhooks implementation.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.task.http.**HttpDispatch**(*url*, *method*, *task_kwargs*, *logger*)
Make task HTTP request and collect the task result.

Parameters

- **url** – The URL to request.
- **method** – HTTP method used. Currently supported methods are *GET* and *POST*.
- **task_kwargs** – Task keyword arguments.
- **logger** – Logger used for user/system feedback.

dispatch()
Dispatch callback and return result.

http_headers

make_request(*url*, *method*, *params*)
Makes an HTTP request and returns the response.

timeout = 5

user_agent = 'celery/2.4.7'

class celery.task.http.**HttpDispatchTask**
Task dispatching to an URL.

Parameters

- **url** – The URL location of the HTTP callback task.
- **method** – Method to use when dispatching the callback. Usually *GET* or *POST*.
- ****kwargs** – Keyword arguments to pass on to the HTTP callback.

url
If this is set, this is used as the default URL for requests. Default is to require the user of the task to supply the url as an argument, as this attribute is intended for subclasses.

method
If this is set, this is the default method used for requests. Default is to require the user of the task to supply the method as an argument, as this attribute is intended for subclasses.

method = None

name = 'celery.task.http.HttpDispatchTask'

run(*url=None*, *method='GET'*, ***kwargs*)

url = None

exception celery.task.http.**InvalidResponseError**
The remote server gave an invalid response.

class celery.task.http.**MutableURL**(*url*)
Object wrapping a Uniform Resource Locator.

Supports editing the query parameter list. You can convert the object back to a string, the query will be properly urlencoded.

Examples

```
>>> url = URL("http://www.google.com:6580/foo/bar?x=3&y=4#foo")
>>> url.query
{'x': '3', 'y': '4'}
>>> str(url)
```

```
'http://www.google.com:6580/foo/bar?y=4&x=3#foo'
>>> url.query["x"] = 10
>>> url.query.update({"George": "Costanza"})
>>> str(url)
'http://www.google.com:6580/foo/bar?y=4&x=10&George=Costanza#foo'
```

exception celery.task.http.**RemoteExecuteError**
The remote task gave a custom error.

class celery.task.http.**URL**(*url*, *dispatcher=None*)
HTTP Callback URL

Supports requesting an URL asynchronously.

Parameters

- **url** – URL to request.
- **dispatcher** – Class used to dispatch the request. By default this is HttpDispatchTask.

dispatcher
alias of HttpDispatchTask

get_async(***kwargs*)

post_async(***kwargs*)

exception celery.task.http.**UnknownStatusError**
The remote server gave an unknown status.

celery.task.http.**extract_response**(*raw_response*)
Extract the response text from a raw JSON response.

celery.task.http.**maybe_utf8**(*value*)
Encode to utf-8, only if the value is Unicode.

celery.task.http.**utf8dict**(*tup*)
With a dict's items() tuple return a new dict with any utf-8 keys/values encoded.

10.12 celery.schedules

- celery.schedules

10.12.1 celery.schedules

Schedules define the intervals at which periodic tasks should run.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

exception celery.schedules.**ParseException**
Raised by crontab_parser when the input can't be parsed.

class celery.schedules.**crontab**(*minute='*', hour='*', day_of_week='*', nowfun=<built-in method now of type object at 0x7feeda7f5c40>*)

A crontab can be used as the *run_every* value of a `PeriodicTask` to add cron-like scheduling.

Like a *cron* job, you can specify units of time of when you would like the task to execute. It is a reasonably complete implementation of cron's features, so it should provide a fair degree of scheduling needs.

You can specify a minute, an hour, and/or a day of the week in any of the following formats:

minute

- A (list of) integers from 0-59 that represent the minutes of an hour of when execution should occur; or
- A string representing a crontab pattern. This may get pretty advanced, like *minute="*/15"* (for every quarter) or *minute="1,13,30-45,50-59/2"*.

hour

- A (list of) integers from 0-23 that represent the hours of a day of when execution should occur; or
- A string representing a crontab pattern. This may get pretty advanced, like *hour="*/3"* (for every three hours) or *hour="0,8-17/2"* (at midnight, and every two hours during office hours).

day_of_week

- A (list of) integers from 0-6, where Sunday = 0 and Saturday = 6, that represent the days of a week that execution should occur.
- A string representing a crontab pattern. This may get pretty advanced, like *day_of_week="mon-fri"* (for weekdays only). (Beware that *day_of_week="*/2"* does not literally mean "every two days", but "every day that is divisible by two"!)

is_due(*last_run_at*)

Returns tuple of two items *(is_due, next_time_to_run)*, where next time to run is in seconds.

See `celery.schedules.schedule.is_due()` for more information.

remaining_estimate(*last_run_at*)

Returns when the periodic task should run next as a timedelta.

class celery.schedules.**crontab_parser**(*max_=60*)

Parser for crontab expressions. Any expression of the form 'groups' (see BNF grammar below) is accepted and expanded to a set of numbers. These numbers represent the units of time that the crontab needs to run on:

```
digit   :: '0'..'9'
dow     :: 'a'..'z'
number  :: digit+ | dow+
steps   :: number
range   :: number ( '-' number ) ?
numspec :: '*' | range
expr    :: numspec ( '/' steps ) ?
groups  :: expr ( ',' expr ) *
```

The parser is a general purpose one, useful for parsing hours, minutes and day_of_week expressions. Example usage:

```
>>> minutes = crontab_parser(60).parse("*/15")
[0, 15, 30, 45]
>>> hours = crontab_parser(24).parse("*/4")
[0, 4, 8, 12, 16, 20]
>>> day_of_week = crontab_parser(7).parse("*")
[0, 1, 2, 3, 4, 5, 6]
```

exception ParseException
Raised by crontab_parser when the input can't be parsed.

crontab_parser.**parse**(*spec*)

celery.schedules.**maybe_schedule**(*s*, *relative=False*)

class celery.schedules.**schedule**(*run_every=None*, *relative=False*)

human_seconds

is_due(*last_run_at*)
Returns tuple of two items *(is_due, next_time_to_run)*, where next time to run is in seconds.

e.g.

- ***(True, 20)*, means the task should be run now, and the next** time to run is in 20 seconds.
- *(False, 12)*, means the task should be run in 12 seconds.

You can override this to decide the interval at runtime, but keep in mind the value of CELERYBEAT_MAX_LOOP_INTERVAL, which decides the maximum number of seconds celerybeat can sleep between re-checking the periodic task intervals. So if you dynamically change the next run at value, and the max interval is set to 5 minutes, it will take 5 minutes for the change to take effect, so you may consider lowering the value of CELERYBEAT_MAX_LOOP_INTERVAL if responsiveness is of importance to you.

relative = False

remaining_estimate(*last_run_at*)
Returns when the periodic task should run next as a timedelta.

seconds

10.13 celery.signals

- celery.signals

10.13.1 celery.signals

See *Signals*.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

10.14 celery.utils.mail

- celery.utils.mail

10.14.1 celery.utils.mail

How task error emails are formatted and sent.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.utils.mail.**ErrorMail**(*task*, ***kwargs*)
Defines how and when task error e-mails should be sent.

Parameters **task** – The task instance that raised the error.

`subject` and `body` are format strings which are passed a context containing the following keys:

•name

Name of the task.

•id

UUID of the task.

•exc

String representation of the exception.

•args

Positional arguments.

•kwargs

Keyword arguments.

•traceback

String representation of the traceback.

•hostname

Worker hostname.

EMAIL_SIGNATURE_SEP = '-- '

body = '\nTask %(name)s with id %(id)s raised exception:\n%(exc)r\n\n\nTask was called with args: %(args)s kwargs: %
Format string used to generate error email content.

error_whitelist = None

format_body(*context*)

format_subject(*context*)

send(*context*, *exc*, *fail_silently=True*)

should_send(*context*, *exc*)
Returns true or false depending on if a task error mail should be sent for this type of error.

subject = ' [celery@%(hostname)s] Error: Task %(name)s (%(id)s): %(exc)s\n '
Format string used to generate error email subjects.

class celery.utils.mail.**Mailer**(*host='localhost'*, *port=0*, *user=None*, *password=None*, *timeout=2*, *use_ssl=False*, *use_tls=False*)

send(*message*)

class celery.utils.mail.**Message**(*to=None*, *sender=None*, *subject=None*, *body=None*, *charset='us-ascii'*)

exception celery.utils.mail.**SendmailWarning**
Problem happened while sending the email message.

10.15 celery.exceptions

- celery.exceptions

10.15.1 celery.exceptions

This module contains Celery-specific exceptions.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

exception celery.exceptions.**AlreadyRegistered**
The task is already registered.

exception celery.exceptions.**CDeprecationWarning**

exception celery.exceptions.**CPendingDeprecationWarning**

exception celery.exceptions.**ImproperlyConfigured**
Celery is somehow improperly configured.

exception celery.exceptions.**MaxRetriesExceededError**
The tasks max restart limit has been exceeded.

exception celery.exceptions.**NotConfigured**
Celery has not been configured, as no config module has been found.

exception celery.exceptions.**NotRegistered**
The task is not registered.

exception celery.exceptions.**QueueNotFound**
Task routed to a queue not in CELERY_QUEUES.

exception celery.exceptions.**RetryTaskError**(*message*, *exc*, **args*, ***kwargs*)
The task is to be retried later.

exception celery.exceptions.**SoftTimeLimitExceeded**
The soft time limit has been exceeded. This exception is raised to give the task a chance to clean up.

exception celery.exceptions.**SystemTerminate**
Signals that the worker should terminate.

exception celery.exceptions.**TaskRevokedError**
The task has been revoked, so no result available.

exception celery.exceptions.**TimeLimitExceeded**
The time limit has been exceeded and the job has been terminated.

exception celery.exceptions.**TimeoutError**
The operation timed out.

exception celery.exceptions.**WorkerLostError**
The worker processing a job has exited prematurely.

10.16 celery.loaders

- celery.loaders

10.16.1 celery.loaders

Loaders define how configuration is read, what happens when workers start, when tasks are executed and so on.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

celery.loaders.**current_loader**(**args*, ***kwargs*)

celery.loaders.**get_loader_cls**(*loader*)
Get loader class by name/alias

celery.loaders.**load_settings**(**args*, ***kwargs*)

10.17 celery.loaders.app

- celery.loaders.app

10.17.1 celery.loaders.app

The default loader used with custom app instances.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.loaders.app.**AppLoader**(*app=None*, ***kwargs*)

on_worker_init()

read_configuration()

10.18 celery.loaders.default

- celery.loaders.default

10.18.1 celery.loaders.default

The default loader used when no custom app has been initialized.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.loaders.default.`**Loader**(*app=None*, ***kwargs*)
The loader used by the default app.

find_module(*module*)

on_worker_init()
Imports modules at worker init so tasks can be registered and used by the worked.

The list of modules to import is taken from the `CELERY_IMPORTS` setting.

read_configuration()
Read configuration from `celeryconfig.py` and configure celery and Django so it can be used by regular Python.

setup_settings(*settingsdict*)

wanted_module_item(*item*)

10.19 celery.loaders.base

- celery.loaders.base

10.19.1 celery.loaders.base

Loader base class.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.loaders.base.`**BaseLoader**(*app=None*, ***kwargs*)
The base class for loaders.

Loaders handles,

•Reading celery client/worker configurations.

•**What happens when a task starts?** See `on_task_init()`.

•**What happens when the worker starts?** See `on_worker_init()`.

•What modules are imported to find tasks?

builtin_modules = frozenset([’celery.task’])

cmdline_config_parser(*args*, *namespace=’celery’*, *re_type=<_sre.SRE_Pattern object at 0x2eb3f08>*, *extra_types={‘json’: <function loads at 0x2f68230>}*, *override_types={‘dict’: ‘json’, ‘list’: ‘json’, ‘tuple’: ‘json’}*)

conf
Loader configuration.

config_from_envvar(*variable_name*, *silent=False*)

config_from_object(*obj*, *silent=False*)

configured = False

error_envvar_not_set = 'The environment variable %r is not set,\nand as such the configuration could not be loade

import_default_modules()

import_from_cwd(*module*, *imp=None*, *package=None*)

import_module(*module*, *package=None*)

import_task_module(*module*)

init_worker()

init_worker_process()

mail

mail_admins(*subject*, *body*, *fail_silently=False*, *sender=None*, *to=None*, *host=None*, *port=None*, *user=None*, *password=None*, *timeout=None*, *use_ssl=False*, *use_tls=False*)

on_process_cleanup()
This method is called after a task is executed.

on_task_init(*task_id*, *task*)
This method is called before a task is executed.

on_worker_init()
This method is called when the worker (**celeryd**) starts.

on_worker_process_init()
This method is called when a child process starts.

override_backends = {}

worker_initialized = False

10.20 celery.registry

- celery.registry

10.20.1 celery.registry

Registry of available tasks.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.registry.**TaskRegistry**

exception NotRegistered
The task is not registered.

TaskRegistry.**filter_types**(*type*)
Return all tasks of a specific type.

TaskRegistry.**periodic**()
Get all periodic task types.

TaskRegistry.**pop**(*key*, **args*)

TaskRegistry.**register**(*task*)
Register a task in the task registry.

The task will be automatically instantiated if not already an instance.

TaskRegistry.**regular**()
Get all regular task types.

TaskRegistry.**unregister**(*name*)
Unregister task by name.

Parameters **name** – name of the task to unregister, or a `celery.task.base.Task` with a valid *name* attribute.

Raises celery.exceptions.NotRegistered if the task has not been registered.

celery.registry.**tasks = {'celery.chord_unlock': <@task: celery.chord_unlock>, 'celery.backend_cleanup': <@task: cel**
Global task registry.

- celery.states
  - States
  - Sets
    - READY_STATES
    - UNREADY_STATES
    - EXCEPTION_STATES
    - PROPAGATE_STATES
    - ALL_STATES
  - Misc.

10.21 celery.states

Built-in Task States.

10.21.1 States

See *Task States*.

10.21.2 Sets

READY_STATES

Set of states meaning the task result is ready (has been executed).

UNREADY_STATES

Set of states meaning the task result is not ready (has not been executed).

EXCEPTION_STATES

Set of states meaning the task returned an exception.

PROPAGATE_STATES

Set of exception states that should propagate exceptions to the user.

ALL_STATES

Set of all possible states.

10.21.3 Misc.

`celery.states.`**`PRECEDENCE`** **= ['SUCCESS', 'FAILURE', None, 'REVOKED', 'STARTED', 'RECEIVED', 'RETRY', 'PE**
State precedence. None represents the precedence of an unknown state. Lower index means higher precedence.

`celery.states.`**`precedence`**`(`*state*`)`
Get the precedence index for state.

Lower index means higher precedence.

class `celery.states.`**`state`**
State is a subclass of `str`, implementing comparison methods adhering to state precedence rules.

10.22 celery.contrib.abortable

- Abortable tasks overview
  - Usage example

10.22.1 Abortable tasks overview

For long-running `Task`'s, it can be desirable to support aborting during execution. Of course, these tasks should be built to support abortion specifically.

The `AbortableTask` serves as a base class for all `Task` objects that should support abortion by producers.

- Producers may invoke the `abort()` method on `AbortableAsyncResult` instances, to request abortion.
- Consumers (workers) should periodically check (and honor!) the `is_aborted()` method at controlled points in their task's `run()` method. The more often, the better.

The necessary intermediate communication is dealt with by the `AbortableTask` implementation.

Usage example

In the consumer:

```
from celery.contrib.abortable import AbortableTask

class MyLongRunningTask(AbortableTask):

    def run(self, **kwargs):
        logger = self.get_logger(**kwargs)
        results = []
        for x in xrange(100):
            # Check after every 5 loops..
            if x % 5 == 0:  # alternatively, check when some timer is due
                if self.is_aborted(**kwargs):
                    # Respect the aborted status and terminate
                    # gracefully
                    logger.warning("Task aborted.")
                    return None
            y = do_something_expensive(x)
            results.append(y)
        logger.info("Task finished.")
        return results
```

In the producer:

```
from myproject.tasks import MyLongRunningTask

def myview(request):

    async_result = MyLongRunningTask.delay()
    # async_result is of type AbortableAsyncResult

    # After 10 seconds, abort the task
    time.sleep(10)
    async_result.abort()

    ...
```

After the *async_result.abort()* call, the task execution is not aborted immediately. In fact, it is not guaranteed to abort at all. Keep checking the *async_result* status, or call *async_result.wait()* to have it block until the task is finished.

Note: In order to abort tasks, there needs to be communication between the producer and the consumer. This is currently implemented through the database backend. Therefore, this class will only work with the database backends.

---

class celery.contrib.abortable.**AbortableAsyncResult**(*task_id*, *backend=None*, *task_name=None*, *app=None*)

Represents a abortable result.

Specifically, this gives the *AsyncResult* a `abort()` method, which sets the state of the underlying Task to *"ABORTED"*.

abort()

Set the state of the task to `ABORTED`.

Abortable tasks monitor their state at regular intervals and terminate execution if so.

Be aware that invoking this method does not guarantee when the task will be aborted (or even if the task will be aborted at all).

is_aborted()
Returns `True` if the task is (being) aborted.

class celery.contrib.abortable.**AbortableTask**
A celery task that serves as a base class for all `Task`'s that support aborting during execution.

All subclasses of `AbortableTask` must call the `is_aborted()` method periodically and act accordingly when the call evaluates to `True`.

classmethod AsyncResult(*task_id*)
Returns the accompanying AbortableAsyncResult instance.

is_aborted(***kwargs*)
Checks against the backend whether this `AbortableAsyncResult` is `ABORTED`.

Always returns `False` in case the *task_id* parameter refers to a regular (non-abortable) `Task`.

Be aware that invoking this method will cause a hit in the backend (for example a database query), so find a good balance between calling it regularly (for responsiveness), but not too often (for performance).

name = 'celery.contrib.abortable.AbortableTask'

10.23 celery.contrib.batches

Collect messages and processes them as a list.

Example

A click counter that flushes the buffer every 100 messages, and every 10 seconds.

```
from celery.task import task
from celery.contrib.batches import Batches

# Flush after 100 messages, or 10 seconds.
@task(base=Batches, flush_every=100, flush_interval=10)
def count_click(requests):
    from collections import Counter
    count = Counter(request.kwargs["url"] for request in requests)
    for url, count in count.items():
        print(">>> Clicks: %s -> %s" % (url, count))
```

Registering the click is done as follows:

```
>>> count_click.delay(url="http://example.com")
```

Warning: For this to work you have to set `CELERYD_PREFETCH_MULTIPLIER` to zero, or some value where the final multiplied value is higher than `flush_every`.
In the future we hope to add the ability to direct batching tasks to a channel with different QoS requirements than the task channel.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

API

class celery.contrib.batches.**Batches**

apply_buffer(*requests*, *args=()*, *kwargs={}*)

debug(*msg*)

execute(*request*, *pool*, *loglevel*, *logfile*)

flush(*requests*)

flush_every = 10
Maximum number of message in buffer.

flush_interval = 30
Timeout in seconds before buffer is flushed anyway.

logger

run(*requests*)

class celery.contrib.batches.**SimpleRequest**(*id*, *name*, *args*, *kwargs*, *delivery_info*, *hostname*)
Pickleable request.

args = ()
positional arguments

delivery_info = None
message delivery information.

classmethod from_request(*request*)

hostname = None
worker node name

id = None
task id

kwargs = {}
keyword arguments

name = None
task name

10.24 celery.contrib.rdb

Remote debugger for Celery tasks running in multiprocessing pool workers. Inspired by http://snippets.dzone.com/posts/show/7248

Usage

```
from celery.contrib import rdb
from celery.decorators import task

@task
def add(x, y):
    result = x + y
    rdb.set_trace()
    return result
```

Environment Variables

CELERY_RDB_HOST
Hostname to bind to. Default is '127.0.01', which means the socket will only be accessible from the local host.

CELERY_RDB_PORT
Base port to bind to. Default is 6899. The debugger will try to find an available port starting from the base port. The selected port will be logged by celeryd.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

`celery.contrib.rdb.`**`set_trace`**(*frame=None*)
Set breakpoint at current location, or a specified frame

`celery.contrib.rdb.`**`debugger`**()
Returns the current debugger instance (if any), or creates a new one.

class `celery.contrib.rdb.`**`Rdb`**(*host='127.0.0.1'*, *port=6899*, *port_search_limit=100*, *port_skew=0*)

10.25 celery.events

- celery.events

10.25.1 celery.events

Events are messages sent for actions happening in the worker (and clients if `CELERY_SEND_TASK_SENT_EVENT` is enabled), used for monitoring purposes.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

`celery.events.`**`Event`**(*type*, *_fields=None*, ***fields*)
Create an event.

An event is a dictionary, the only required field is `type`.

class `celery.events.`**`EventDispatcher`**(*connection=None*, *hostname=None*, *enabled=True*, *channel=None*, *buffer_while_offline=True*, *app=None*, *serializer=None*)
Send events as messages.

Parameters

- **connection** – Connection to the broker.
- **hostname** – Hostname to identify ourselves as, by default uses the hostname returned by `socket.gethostname()`.
- **enabled** – Set to `False` to not actually publish any events, making `send()` a noop operation.
- **channel** – Can be used instead of *connection* to specify an exact channel to use when sending events.
- **buffer_while_offline** – If enabled events will be buffered while the connection is down. `flush()` must be called as soon as the connection is re-established.

You need to `close()` this after use.

close()
Close the event dispatcher.

copy_buffer(*other*)

disable()

enable()

flush()

send(*type*, ***fields*)
Send event.

Parameters

- **type** – Kind of event.
- ****fields** – Event arguments.

class celery.events.**EventReceiver**(*connection*, *handlers=None*, *routing_key='#'*, *node_id=None*, *app=None*)
Capture events.

Parameters

- **connection** – Connection to the broker.
- **handlers** – Event handlers.

`handlers` is a dict of event types and their handlers, the special handler *"*"* captures all events that doesn't have a handler.

capture(*limit=None*, *timeout=None*, *wakeup=True*)
Open up a consumer capturing events.

This has to run in the main process, and it will never stop unless forced via `KeyboardInterrupt` or `SystemExit`.

consumer(**args*, ***kwds*)
Create event consumer.

> **Warning:** This creates a new channel that needs to be closed by calling *consumer.channel.close()*.

drain_events(*limit=None*, *timeout=None*)

handlers = {}

itercapture(*limit=None*, *timeout=None*, *wakeup=True*)

process(*type*, *event*)
Process the received event by dispatching it to the appropriate handler.

wakeup_workers(*channel=None*)

class celery.events.**Events**(*app=None*)

Dispatcher(*connection=None*, *hostname=None*, *enabled=True*, *channel=None*, *buffer_while_offline=True*)

Receiver(*connection*, *handlers=None*, *routing_key='#'*, *node_id=None*)

State()

default_dispatcher(**args*, ***kwds*)

10.26 celery.events.state

- celery.events.state

10.26.1 celery.events.state

This module implements a datastructure used to keep track of the state of a cluster of workers and the tasks it is working on (by consuming events).

For every event consumed the state is updated, so the state represents the state of the cluster at the time of the last event.

Snapshots (`celery.events.snapshot`) can be used to take "pictures" of this state at regular intervals to e.g. store that in a database.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.events.state.`**Element**
Base class for worker state elements.

`celery.events.state.`**HEARTBEAT_EXPIRE = 150**
Hartbeat expiry time in seconds. The worker will be considered offline if no heartbeat is received within this time. Default is 2:30 minutes.

class `celery.events.state.`**State**(*callback=None*, *max_workers_in_memory=5000*, *max_tasks_in_memory=10000*)
Records clusters state.

alive_workers()
Returns a list of (seemingly) alive workers.

clear(*ready=True*)

clear_tasks(*ready=True*)

event(*event*)

event_count = 0

freeze_while(*fun*, **args*, ***kwargs*)

get_or_create_task(*uuid*)
Get or create task by uuid.

get_or_create_worker(*hostname*, ***kwargs*)
Get or create worker by hostname.

itertasks(*limit=None*)

task_count = 0

task_event(*type*, *fields*)
Process task event.

task_types()
Returns a list of all seen task types.

tasks_by_timestamp(*limit=None*)
Get tasks by timestamp.

Returns a list of *(uuid, task)* tuples.

tasks_by_type(*name*, *limit=None*)
Get all tasks by type.

Returns a list of *(uuid, task)* tuples.

tasks_by_worker(*hostname*, *limit=None*)
Get all tasks by worker.

Returns a list of *(uuid, task)* tuples.

worker_event(*type*, *fields*)
Process worker event.

class celery.events.state.**Task**(***fields*)
Task State.

info(*fields=None*, *extra=[]*)
Information about this task suitable for on-screen display.

merge(*state*, *timestamp*, *fields*)
Merge with out of order event.

merge_rules = {'RECEIVED': ('name', 'args', 'kwargs', 'retries', 'eta', 'expires')}
How to merge out of order events. Disorder is detected by logical ordering (e.g. task-received must have happened before a task-failed event).

A merge rule consists of a state and a list of fields to keep from that state. `(RECEIVED, ("name", "args")`, means the name and args fields are always taken from the RECEIVED state, and any values for these fields received before or after is simply ignored.

on_failed(*timestamp=None*, ***fields*)
Callback for the `task-failed` event.

on_received(*timestamp=None*, ***fields*)
Callback for the `task-received` event.

on_retried(*timestamp=None*, ***fields*)
Callback for the `task-retried` event.

on_revoked(*timestamp=None*, ***fields*)
Callback for the `task-revoked` event.

on_sent(*timestamp=None*, ***fields*)
Callback for the `task-sent` event.

on_started(*timestamp=None*, ***fields*)
Callback for the `task-started` event.

on_succeeded(*timestamp=None*, ***fields*)
Callback for the `task-succeeded` event.

on_unknown_event(*type*, *timestamp=None*, ***fields*)

ready

update(*state*, *timestamp*, *fields*)
Update state from new event.

Parameters

- **state** – State from event.
- **timestamp** – Timestamp from event.
- **fields** – Event data.

class celery.events.state.**Worker**(***fields*)
Worker State.

alive

heartbeat_max = 4

on_heartbeat(*timestamp=None*, ***kwargs*)
Callback for the *worker-heartbeat* event.

on_offline(***kwargs*)
Callback for the *worker-offline* event.

on_online(*timestamp=None*, ***kwargs*)
Callback for the *worker-online* event.

10.27 celery.apps.worker

class celery.apps.worker.**Worker**(*concurrency=None*, *loglevel=None*, *logfile=None*, *hostname=None*, *discard=False*, *run_clockservice=False*, *schedule=None*, *task_time_limit=None*, *task_soft_time_limit=None*, *max_tasks_per_child=None*, *queues=None*, *events=None*, *db=None*, *include=None*, *app=None*, *pidfile=None*, *redirect_stdouts=None*, *redirect_stdouts_level=None*, *autoscale=None*, *scheduler_cls=None*, *pool=None*, ***kwargs*)

class WorkController(*concurrency=None*, *logfile=None*, *loglevel=None*, *send_events=None*, *hostname=None*, *ready_callback=<function noop at 0x2ef6848>*, *embed_clockservice=False*, *pool_cls=None*, *consumer_cls=None*, *mediator_cls=None*, *eta_scheduler_cls=None*, *schedule_filename=None*, *task_time_limit=None*, *task_soft_time_limit=None*, *max_tasks_per_child=None*, *pool_putlocks=None*, *db=None*, *prefetch_multiplier=None*, *eta_scheduler_precision=None*, *disable_rate_limits=None*, *autoscale=None*, *autoscaler_cls=None*, *scheduler_cls=None*, *app=None*)
Unmanaged worker instance.

CLOSE = 2

RUN = 1

TERMINATE = 3

concurrency = None

consumer = None

embed_clockservice = None

logfile = None

logger = None

loglevel = 40

mediator = None

on_timer_error(*exc_info*)

on_timer_tick(*delay*)

pool = None

process_task(*request*)
Process task by sending it to the pool of workers.

ready_queue = None

send_events = False

start()
Starts the workers main loop.

stop(*in_sighandler=False*)
Graceful shutdown of the worker server.

terminate(*in_sighandler=False*)
Not so graceful shutdown of the worker server.

Worker.**die**(*msg*, *exitcode=1*)

Worker.**extra_info**()

Worker.**init_loader**()

Worker.**init_queues**()

Worker.**install_platform_tweaks**(*worker*)
Install platform specific tweaks and workarounds.

Worker.**on_consumer_ready**(*consumer*)

Worker.**osx_proxy_detection_workaround**()
See http://github.com/ask/celery/issues#issue/161

Worker.**purge_messages**()

Worker.**redirect_stdouts_to_logger**()

Worker.**run**()

Worker.**run_worker**()

Worker.**set_process_status**(*info*)

Worker.**startup_info**()

Worker.**tasklist**(*include_builtins=True*)

Worker.**worker_init**()

celery.apps.worker.**cpu_count**()

celery.apps.worker.**get_process_name**()

celery.apps.worker.**install_HUP_not_supported_handler**(*worker*)

celery.apps.worker.**install_cry_handler**(*logger*)

celery.apps.worker.**install_rdb_handler**(*envvar='CELERY_RDBSIG'*)

celery.apps.worker.**install_worker_int_again_handler**(*worker*)

celery.apps.worker.**install_worker_int_handler**(*worker*)

celery.apps.worker.**install_worker_restart_handler**(*worker*)

celery.apps.worker.**install_worker_term_handler**(*worker*)

10.28 celery.apps.beat

class celery.apps.beat.**Beat**(*loglevel=None*, *logfile=None*, *schedule=None*, *max_interval=None*, *scheduler_cls=None*, *app=None*, *socket_timeout=30*, *redirect_stdouts=None*, *redirect_stdouts_level=None*, *pidfile=None*, ***kwargs*)

class Service(*logger=None*, *max_interval=None*, *schedule_filename=None*, *scheduler_cls=None*, *app=None*)

get_scheduler(*lazy=False*)

scheduler

scheduler_cls
alias of PersistentScheduler

start(*embedded_process=False*)

stop(*wait=False*)

sync()

Beat.**init_loader**()

Beat.**install_sync_handler**(*beat*)
Install a *SIGTERM* + *SIGINT* handler that saves the celerybeat schedule.

Beat.**run**()

Beat.**set_process_title**()

Beat.**setup_logging**()

Beat.**start_scheduler**(*logger=None*)

Beat.**startup_info**(*beat*)

10.29 celery.bin.base

class celery.bin.base.**Command**(*app=None*, *get_app=None*)
Base class for command line applications.

Parameters

- **app** – The current app.
- **get_app** – Callable returning the current app if no app provided.

Parser
alias of OptionParser

args = ''
Arg list used in help.

create_parser(*prog_name*)

enable_config_from_cmdline = False
Enable if the application should support config from the cmdline.

execute_from_commandline(*argv=None*)
Execute application from command line.

Parameters **argv** – The list of command line arguments. Defaults to `sys.argv`.

get_cls_by_name(*name*)

get_options()
Get supported command line options.

handle_argv(*prog_name*, *argv*)
Parses command line arguments from `argv` and dispatches to `run()`.

Parameters

- **prog_name** – The program name (`argv[0]`).
- **argv** – Command arguments.

Exits with an error message if `supports_args` is disabled and `argv` contains positional arguments.

namespace = 'celery'
Default configuration namespace.

option_list = ()
List of options (without preload options).

parse_options(*prog_name*, *arguments*)
Parse the available options.

parse_preload_options(*args*)

preload_options = (<Option at 0x81083b0: –app>, <Option at 0x8108c20: -b/–broker>, <Option at 0x8108d88: –load
List of options to parse before parsing other options.

prepare_preload_options(*options*)
Optional handler to do additional processing of preload options.

Configuration must not have been initialized until after this is called.

process_cmdline_config(*argv*)

run(**args*, ***options*)
This is the body of the command called by `handle_argv()`.

setup_app_from_commandline(*argv*)

supports_args = True
If false the parser will raise an exception if positional args are provided.

usage()
Returns the command-line usage string for this app.

version = '2.4.7'
Application version.

`celery.bin.base.`**daemon_options**(*default_pidfile=None*, *default_logfile=None*)

10.30 celery.bin.celeryd

celeryd

-c, --concurrency
Number of child processes processing the queue. The default is the number of CPUs available on your system.

-f, --logfile
Path to log file. If no logfile is specified, *stderr* is used.

-l, --loglevel
Logging level, choose between *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*, or *FATAL*.

-n, --hostname
Set custom hostname.

-B, --beat
Also run the *celerybeat* periodic task scheduler. Please note that there must only be one instance of this service.

-Q, --queues
List of queues to enable for this worker, separated by comma. By default all configured queues are enabled. Example: *-Q video,image*

-I, --include
Comma separated list of additional modules to import. Example: -I foo.tasks,bar.tasks

-s, --schedule
Path to the schedule database if running with the *-B* option. Defaults to *celerybeat-schedule*. The extension ".db" will be appended to the filename.

--scheduler
Scheduler class to use. Default is celery.beat.PersistentScheduler

-E, --events
Send events that can be captured by monitors like *celerymon*.

--purge, --discard
Discard all waiting tasks before the daemon is started. **WARNING**: This is unrecoverable, and the tasks will be deleted from the messaging server.

--time-limit
Enables a hard time limit (in seconds) for tasks.

--soft-time-limit
Enables a soft time limit (in seconds) for tasks.

--maxtasksperchild
Maximum number of tasks a pool worker can execute before it's terminated and replaced by a new worker.

class celery.bin.celeryd.**WorkerCommand**(*app=None*, *get_app=None*)

enable_config_from_cmdline = True

get_options()

namespace = 'celeryd'

run(**args*, ***kwargs*)

supports_args = False

celery.bin.celeryd.**main**()

celery.bin.celeryd.**windows_main**()

10.31 celery.bin.celerybeat

celerybeat

-s, --schedule
Path to the schedule database. Defaults to *celerybeat-schedule*. The extension ".db" will be appended to the filename.

-S, --scheduler
Scheduler class to use. Default is celery.beat.PersistentScheduler

-f, --logfile
Path to log file. If no logfile is specified, *stderr* is used.

-l, --loglevel
Logging level, choose between *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*, or *FATAL*.

class celery.bin.celerybeat.**BeatCommand**(*app=None*, *get_app=None*)

get_options()

preload_options = (<Option at 0x81083b0: –app>, <Option at 0x8108c20: -b/–broker>, <Option at 0x8108d88: –load

prepare_preload_options(*options*)

run(*detach=False*, *logfile=None*, *pidfile=None*, *uid=None*, *gid=None*, *umask=None*, *working_directory=None*, ***kwargs*)

supports_args = False

celery.bin.celerybeat.**main**()

10.32 celery.bin.celeryev

class celery.bin.celeryev.**EvCommand**(*app=None*, *get_app=None*)

get_options()

preload_options = (<Option at 0x81083b0: –app>, <Option at 0x8108c20: -b/–broker>, <Option at 0x8108d88: –load

prepare_preload_options(*options*)

run(*dump=False*, *camera=None*, *frequency=1.0*, *maxrate=None*, *loglevel='INFO'*, *logfile=None*, *prog_name='celeryev'*, *pidfile=None*, *uid=None*, *gid=None*, *umask=None*, *working_directory=None*, *detach=False*, ***kwargs*)

run_evcam(*camera*, *logfile=None*, *pidfile=None*, *uid=None*, *gid=None*, *umask=None*, *working_directory=None*, *detach=False*, ***kwargs*)

run_evdump()

run_evtop()

set_process_status(*prog*, *info=''*)

supports_args = False

celery.bin.celeryev.**main**()

10.33 celeryctl - celery.bin.celeryctl

class celery.bin.celeryctl.**Command**(*app=None*, *no_color=False*)

args = ''

create_parser(*prog_name*, *command*)

error(*s*)

help = ''

option_list = (<Option at 0x81083b0: –app>, <Option at 0x8108c20: -b/–broker>, <Option at 0x8108d88: –loader>, <

out(*s*, *fh=<open file '<stdout>', mode 'w' at 0x7feedd811150>*)

prettify(*n*)

prettify_dict_ok_error(*n*)

prettify_list(*n*)

run(**args*, ***kwargs*)

run_from_argv(*prog_name*, *argv*)

usage(*command*)

version = '2.4.7'

exception celery.bin.celeryctl.**Error**

class celery.bin.celeryctl.**apply**(*app=None*, *no_color=False*)

args = '<task_name>'

option_list = (<Option at 0x81083b0: –app>, <Option at 0x8108c20: -b/–broker>, <Option at 0x8108d88: –loader>, <

run(*name*, **_*, ***kw*)

class celery.bin.celeryctl.**celeryctl**(*app=None*, *get_app=None*)

commands = {'status': <class 'celery.bin.celeryctl.status'>, 'help': <class 'celery.bin.celeryctl.help'>, 'inspect': <class 'cel

execute(*command*, *argv=None*)

handle_argv(*prog_name*, *argv*)

remove_options_at_beginning(*argv*, *index=0*)

celery.bin.celeryctl.**command**(*fun*, *name=None*)

class celery.bin.celeryctl.**help**(*app=None*, *no_color=False*)

run(**args*, ***kwargs*)

usage(*command*)

celery.bin.celeryctl.**indent**(*s*, *n=4*)

class celery.bin.celeryctl.**inspect**(*app=None*, *no_color=False*)

choices = {'scheduled': 1.0, 'reserved': 1.0, 'registered': 1.0, 'active': 1.0, 'cancel_consumer': 1.0, 'active_queues': 1.0,

option_list = (<Option at 0x81083b0: –app>, <Option at 0x8108c20: -b/–broker>, <Option at 0x8108d88: –loader>, <

run(**args*, ***kwargs*)

say(*direction*, *title*, *body=''*)

show_body = True

usage(*command*)

class celery.bin.celeryctl.**list_**(*app=None*, *no_color=False*)

args = '<bindings>'

list_bindings(*channel*)

run(*what*, *_*, ***kw*)

celery.bin.celeryctl.**main**()

celery.bin.celeryctl.**pluralize**(*n*, *text*, *suffix='s'*)

class celery.bin.celeryctl.**purge**(*app=None*, *no_color=False*)

run(**args*, ***kwargs*)

class celery.bin.celeryctl.**result**(*app=None*, *no_color=False*)

args = '<task_id>'

option_list = (<Option at 0x81083b0: –app>, <Option at 0x8108c20: -b/–broker>, <Option at 0x8108d88: –loader>, <

run(*task_id*, **args*, ***kwargs*)

class celery.bin.celeryctl.**status**(*app=None*, *no_color=False*)

option_list = (<Option at 0x81083b0: –app>, <Option at 0x8108c20: -b/–broker>, <Option at 0x8108d88: –loader>, <

run(**args*, ***kwargs*)

10.34 celery.bin.camqadm

camqadm

class celery.bin.camqadm.**AMQPAdmin**(**args*, ***kwargs*)
The celery **camqadm** utility.

connect(*conn=None*)

run()

say(*m*)

class celery.bin.camqadm.**AMQPAdminCommand**(*app=None*, *get_app=None*)

run(**args*, ***options*)

class celery.bin.camqadm.**AMQShell**(**args*, ***kwargs*)
AMQP API Shell.

Parameters

- **connect** – Function used to connect to the server, must return connection object.
- **silent** – If `True`, the commands won't have annoying output not relevant when running in non-shell mode.

amqp
Mapping of AMQP API commands and their `Spec`.

amqp = {'queue.declare': <celery.bin.camqadm.Spec object at 0x5b009d0>, 'queue.purge': <celery.bin.camqadm.Spec ob

builtins = {'exit': 'do_exit', 'EOF': 'do_exit', 'help': 'do_help'}

chan = None

completenames(*text*, **ignored*)
Return all commands starting with *text*, for tab-completion.

conn = None

counter = 1

default(*line*)

dispatch(*cmd*, *argline*)
Dispatch and execute the command.

Lookup order is: `builtins` -> `amqp`.

display_command_help(*cmd*, *short=False*)

do_exit(**args*)
The *"exit"* command.

do_help(**args*)

get_amqp_api_command(*cmd*, *arglist*)
With a command name and a list of arguments, convert the arguments to Python values and find the corresponding method on the AMQP channel object.

Returns tuple of *(method, processed_args)*.

Example:

```
>>> get_amqp_api_command("queue.delete", ["pobox", "yes", "no"])
(<bound method Channel.queue_delete of
 <amqplib.client_0_8.channel.Channel object at 0x...>>,
 ('testfoo', True, False))
```

get_names()

identchars = '.'

inc_counter = <method-wrapper 'next' of itertools.count object at 0x593b5a8>

needs_reconnect = False

onecmd(*line*)
Parse line and execute command.

parseline(*line*)
Parse input line.

Returns tuple of three items: *(command_name, arglist, original_line)*

E.g:

```
>>> parseline("queue.delete A 'B' C")
("queue.delete", "A 'B' C", "queue.delete A 'B' C")
```

prompt

prompt_fmt = '%d> '

respond(*retval*)
What to do with the return value of a command.

say(*m*)
Say something to the user. Disabled if `silent`.

class celery.bin.camqadm.**Spec**(**args*, ***kwargs*)
AMQP Command specification.

Used to convert arguments to Python values and display various help and tooltips.

Parameters

- **args** – see `args`.
- **returns** – see `returns`.

coerce(*index*, *value*)
Coerce value for argument at index.

E.g. if `args` is *[("is_active", bool)]*:

```
>>> coerce(0, "False")
False
```

format_arg(*name*, *type*, *default_value=None*)

format_response(*response*)
Format the return value of this command in a human-friendly way.

format_signature()

str_args_to_python(*arglist*)
Process list of string arguments to values according to spec.

e.g:

```
>>> spec = Spec([("queue", str), ("if_unused", bool)])
>>> spec.str_args_to_python("pobox", "true")
("pobox", True)
```

celery.bin.camqadm.**camqadm**(**args*, ***options*)

celery.bin.camqadm.**dump_message**(*message*)

celery.bin.camqadm.**format_declare_queue**(*ret*)

celery.bin.camqadm.**main**()

celery.bin.camqadm.**say**(*m*)

10.35 celery.bin.celeryd_multi

- Examples

10.35.1 Examples

```
# Single worker with explicit name and events enabled.
$ celeryd-multi start Leslie -E

# Pidfiles and logfiles are stored in the current directory
# by default.  Use --pidfile and --logfile argument to change
# this.  The abbreviation %n will be expanded to the current
# node name.
$ celeryd-multi start Leslie -E --pidfile=/var/run/celery/%n.pid
                                --logfile=/var/log/celery/%n.log


# You need to add the same arguments when you restart,
# as these are not persisted anywhere.
$ celeryd-multi restart Leslie -E --pidfile=/var/run/celery/%n.pid
                                  --logfile=/var/run/celery/%n.log

# To stop the node, you need to specify the same pidfile.
$ celeryd-multi stop Leslie --pidfile=/var/run/celery/%n.pid

# 3 workers, with 3 processes each
$ celeryd-multi start 3 -c 3
celeryd -n celeryd1.myhost -c 3
celeryd -n celeryd2.myhost -c 3
celeryd- n celeryd3.myhost -c 3

# start 3 named workers
$ celeryd-multi start image video data -c 3
celeryd -n image.myhost -c 3
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3

# specify custom hostname
$ celeryd-multi start 2 -n worker.example.com -c 3
celeryd -n celeryd1.worker.example.com -c 3
celeryd -n celeryd2.worker.example.com -c 3

# Advanced example starting 10 workers in the background:
#   * Three of the workers processes the images and video queue
#   * Two of the workers processes the data queue with loglevel DEBUG
#   * the rest processes the default' queue.
$ celeryd-multi start 10 -l INFO -Q:1-3 images,video -Q:4,5:data
    -Q default -L:4,5 DEBUG

# You can show the commands necessary to start the workers with
# the "show" command:
$ celeryd-multi show 10 -l INFO -Q:1-3 images,video -Q:4,5:data
    -Q default -L:4,5 DEBUG

# Additional options are added to each celeryd',
# but you can also modify the options for ranges of, or specific workers

# 3 workers: Two with 3 processes, and one with 10 processes.
$ celeryd-multi start 3 -c 3 -c:1 10
celeryd -n celeryd1.myhost -c 10
celeryd -n celeryd2.myhost -c 3
celeryd -n celeryd3.myhost -c 3
```

```
# can also specify options for named workers
$ celeryd-multi start image video data -c 3 -c:image 10
celeryd -n image.myhost -c 10
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3

# ranges and lists of workers in options is also allowed:
# (-c:1-3 can also be written as -c:1,2,3)
$ celeryd-multi start 5 -c 3  -c:1-3 10
celeryd -n celeryd1.myhost -c 10
celeryd -n celeryd2.myhost -c 10
celeryd -n celeryd3.myhost -c 10
celeryd -n celeryd4.myhost -c 3
celeryd -n celeryd5.myhost -c 3

# lists also works with named workers
$ celeryd-multi start foo bar baz xuzzy -c 3 -c:foo,bar,baz 10
celeryd -n foo.myhost -c 10
celeryd -n bar.myhost -c 10
celeryd -n baz.myhost -c 10
celeryd -n xuzzy.myhost -c 3
```

class celery.bin.celeryd_multi.**MultiTool**(*env=None*, *fh=None*)

error(*msg=None*)

execute_from_commandline(*argv*, *cmd='celeryd'*)

expand(*argv*, *cmd=None*)

get(*argv*, *cmd*)

getpids(*p*, *cmd*, *callback=None*)

help(*argv*, *cmd=None*)

info(*msg*, *newline=True*)

kill(*argv*, *cmd*)

names(*argv*, *cmd*)

node_alive(*pid*)

note(*msg*, *newline=True*)

restart(*argv*, *cmd*)

retcode = 0

say(*msg*)

show(*argv*, *cmd*)

shutdown_nodes(*nodes*, *sig=15*, *retry=None*, *callback=None*)

signal_node(*nodename*, *pid*, *sig*)

splash()

start(*argv*, *cmd*)

stop(*argv*, *cmd*)

stop_verify(*argv*, *cmd*)

usage()

waitexec(*argv*, *path='/home/docs/checkouts/readthedocs.org/user_builds/celery/envs/2.4-archived/bin/python'*)

with_detacher_default_options(*p*)

class celery.bin.celeryd_multi.**NamespacedOptionParser**(*args*)

add_option(*name*, *value*, *short=False*, *ns=None*)

optmerge(*ns*, *defaults=None*)

parse()

process_long_opt(*arg*, *value=None*)

process_short_opt(*arg*, *value=None*)

celery.bin.celeryd_multi.**abbreviations**(*map*)

celery.bin.celeryd_multi.**findsig**(*args*, *default=15*)

celery.bin.celeryd_multi.**format_opt**(*opt*, *value*)

celery.bin.celeryd_multi.**main**()

celery.bin.celeryd_multi.**multi_args**(*p*, *cmd='celeryd'*, *append=''*, *prefix=''*, *suffix=''*)

celery.bin.celeryd_multi.**parse_ns_range**(*ns*, *ranges=False*)

celery.bin.celeryd_multi.**quote**(*v*)

celery.bin.celeryd_multi.**say**(*m*, *newline=True*)

CHAPTER 11

Internals

Release 2.4

Date February 04, 2014

11.1 Celery Deprecation Timeline

- Removals for version 2.0

11.1.1 Removals for version 2.0

- The following settings will be removed:

Setting name	Replace with
CELERY_AMQP_CONSUMER_QUEUES	*CELERY_QUEUES*
CELERY_AMQP_CONSUMER_QUEUES	*CELERY_QUEUES*
CELERY_AMQP_EXCHANGE	*CELERY_DEFAULT_EXCHANGE*
CELERY_AMQP_EXCHANGE_TYPE	*CELERY_DEFAULT_AMQP_EXCHANGE_TYPE*
CELERY_AMQP_CONSUMER_ROUTING_KEY	*CELERY_QUEUES*
CELERY_AMQP_PUBLISHER_ROUTING_KEY	*CELERY_DEFAULT_ROUTING_KEY*

- `CELERY_LOADER` definitions without class name.

  E.g. *celery.loaders.default*, needs to include the class name: *celery.loaders.default.Loader*.

- **`TaskSet.run()`. Use `celery.task.base.TaskSet.apply_async()`** instead.
- The module `celery.task.rest`; use `celery.task.http` instead.

11.2 Internals: The worker

- Introduction
- Data structures
  - ready_queue
  - eta_schedule
- Components
  - Consumer
  - ScheduleController
  - Mediator
  - TaskPool

11.2.1 Introduction

The worker consists of 4 main components: the consumer, the scheduler, the mediator and the task pool. All these components runs in parallel working with two data structures: the ready queue and the ETA schedule.

11.2.2 Data structures

ready_queue

The ready queue is either an instance of `Queue.Queue`, or `celery.buckets.TaskBucket`. The latter if rate limiting is enabled.

eta_schedule

The ETA schedule is a heap queue sorted by time.

11.2.3 Components

Consumer

Receives messages from the broker using Kombu.

When a message is received it's converted into a `celery.worker.job.TaskRequest` object.

Tasks with an ETA are entered into the *eta_schedule*, messages that can be immediately processed are moved directly to the *ready_queue*.

ScheduleController

The schedule controller is running the *eta_schedule*. If the scheduled tasks eta has passed it is moved to the *ready_queue*, otherwise the thread sleeps until the eta is met (remember that the schedule is sorted by time).

Mediator

The mediator simply moves tasks in the *ready_queue* over to the task pool for execution using `celery.worker.job.TaskRequest.execute_using_pool()`.

TaskPool

This is a slightly modified `multiprocessing.Pool`. It mostly works the same way, except it makes sure all of the workers are running at all times. If a worker is missing, it replaces it with a new one.

11.3 Task Messages

- Message format
- Extensions
- Example message
- Serialization

11.3.1 Message format

- **task** *string*

  Name of the task. **required**

- **id** *string*

  Unique id of the task (UUID). **required**

- **args** *list*

  List of arguments. Will be an empty list if not provided.

- **kwargs** *dictionary*

  Dictionary of keyword arguments. Will be an empty dictionary if not provided.

- **retries** *int*

  Current number of times this task has been retried. Defaults to *0* if not specified.

- **eta** *string* (ISO 8601)

  Estimated time of arrival. This is the date and time in ISO 8601 format. If not provided the message is not scheduled, but will be executed asap.

- **expires (introduced after v2.0.2)** *string* (ISO 8601)

  Expiration date. This is the date and time in ISO 8601 format. If not provided the message will never expire. The message will be expired when the message is received and the expiration date has been exceeded.

11.3.2 Extensions

Extensions are additional keys in the message body that the worker may or may not support. If the worker finds an extension key it doesn't support it should optimally reject the message so another worker gets a chance to process it.

- taskset_id

  The taskset this task is part of.

- chord

  Siginifies that this task is one of the header parts of a chord. The value of this key is the body of the cord that should be executed when all of the tasks in the header has returned.

11.3.3 Example message

This is an example invocation of the *celery.task.PingTask* task in JSON format:

```
{"id": "4cc7438e-afd4-4f8f-a2f3-f46567e7ca77",
 "task": "celery.task.PingTask",
 "args": [],
 "kwargs": {},
 "retries": 0,
 "eta": "2009-11-17T12:30:56.527191"}
```

11.3.4 Serialization

Several types of serialization formats are supported using the *content_type* message header.

The MIME-types supported by default are shown in the following table.

Scheme	MIME Type
json	application/json
yaml	application/x-yaml
pickle	application/x-python-serialize
msgpack	application/x-msgpack

11.4 "The Big Instance" Refactor

The *app* branch is a work-in-progress to remove the use of a global configuration in Celery.

Celery can now be instantiated, which means several instances of Celery may exist in the same process space. Also, large parts can be customized without resorting to monkey patching.

11.4.1 Examples

Creating a Celery instance:

```
>>> from celery import Celery
>>> celery = Celery()
>>> celery.config_from_object("celeryconfig")
>>> celery.config_from_envvar("CELERY_CONFIG_MODULE")
```

Creating tasks:

```
@celery.task()
def add(x, y):
    return x + y
```

Creating custom Task subclasses:

```
Task = celery.create_task_cls()

class DebugTask(Task):
    abstract = True

    def on_failure(self, \*args, \*\*kwargs):
        import pdb
        pdb.set_trace()

@celery.task(base=DebugTask)
def add(x, y):
    return x + y
```

Starting a worker:

```
worker = celery.Worker(loglevel="INFO")
```

Getting access to the configuration:

```
celery.conf.CELERY_ALWAYS_EAGER = True
celery.conf["CELERY_ALWAYS_EAGER"] = True
```

Controlling workers:

```
>>> celery.control.inspect().active()
>>> celery.control.rate_limit(add.name, "100/m")
>>> celery.control.broadcast("shutdown")
>>> celery.control.discard_all()
```

Other interesting attributes:

```
# Establish broker connection.
>>> celery.broker_connection()

# AMQP Specific features.
>>> celery.amqp
>>> celery.amqp.Router
>>> celery.amqp.get_queues()
>>> celery.amqp.get_task_consumer()

# Loader
>>> celery.loader

# Default backend
>>> celery.backend
```

As you can probably see, this really opens up another dimension of customization abilities.

11.4.2 Deprecations

- celery.task.ping celery.task.PingTask

  Inferior to the ping remote control command. Will be removed in Celery 2.3.

11.4.3 Removed deprecations

- ***celery.utils.timedelta_seconds*** Use: `celery.utils.timeutils.timedelta_seconds()`

- ***celery.utils.defaultdict*** Use: `celery.utils.compat.defaultdict()`
- ***celery.utils.all*** Use: `celery.utils.compat.all()`
- ***celery.task.apply_async*** Use app.send_task
- ***celery.task.tasks*** Use `celery.registry.tasks`

11.4.4 Aliases (Pending deprecation)

- **celery.task.base**
  - .Task -> {app.create_task_cls}
- **celery.task.sets**
  - .TaskSet -> {app.TaskSet}
- **celery.decorators / celery.task**
  - .task -> {app.task}
- **celery.execute**
  - .apply_async -> {task.apply_async}
  - .apply -> {task.apply}
  - .send_task -> {app.send_task}
  - .delay_task -> no alternative
- **celery.log**
  - .get_default_logger -> {app.log.get_default_logger}
  - .setup_logger -> {app.log.setup_logger}
  - .get_task_logger -> {app.log.get_task_logger}
  - .setup_task_logger -> {app.log.setup_task_logger}
  - .setup_logging_subsystem -> {app.log.setup_logging_subsystem}
  - .redirect_stdouts_to_logger -> {app.log.redirect_stdouts_to_logger}
- **celery.messaging**
  - .establish_connection -> {app.broker_connection}
  - .with_connection -> {app.with_connection}
  - .get_consumer_set -> {app.amqp.get_task_consumer}
  - .TaskPublisher -> {app.amqp.TaskPublisher}
  - .TaskConsumer -> {app.amqp.TaskConsumer}
  - .ConsumerSet -> {app.amqp.ConsumerSet}
- celery.conf.* -> {app.conf}

  NOTE: All configuration keys are now named the same as in the configuration. So the key "CELERY_ALWAYS_EAGER" is accessed as:

  ```
  >>> app.conf.CELERY_ALWAYS_EAGER
  ```

  instead of:

```
>>> from celery import conf
>>> conf.ALWAYS_EAGER
```

  - .get_queues -> {app.amqp.get_queues}

- **celery.task.control**
  - .broadcast -> {app.control.broadcast}
  - .rate_limit -> {app.control.rate_limit}
  - .ping -> {app.control.ping}
  - .revoke -> {app.control.revoke}
  - .discard_all -> {app.control.discard_all}
  - .inspect -> {app.control.inspect}
- **celery.utils.info**
  - .humanize_seconds -> celery.utils.timeutils.humanize_seconds
  - .textindent -> celery.utils.textindent
  - .get_broker_info -> {app.amqp.get_broker_info}
  - .format_broker_info -> {app.amqp.format_broker_info}
  - .format_queues -> {app.amqp.format_queues}

11.4.5 Default App Usage

To be backward compatible, it must be possible to use all the classes/functions without passing an explicit app instance.

This is achieved by having all app-dependent objects use `default_app` if the app instance is missing.

```
from celery.app import app_or_default

class SomeClass(object):

    def __init__(self, app=None):
        self.app = app_or_default(app)
```

The problem with this approach is that there is a chance that the app instance is lost along the way, and everything seems to be working normally. Testing app instance leaks is hard. The environment variable `CELERY_TRACE_APP` can be used, when this is enabled `celery.app.app_or_default()` will raise an exception whenever it has to go back to the default app instance.

App Dependency Tree

- **{app}**
  - celery.loaders.base.BaseLoader
  - celery.backends.base.BaseBackend
  - **{app.TaskSet}**
    - celery.task.sets.TaskSet (app.TaskSet)
  - **[app.TaskSetResult]**

        * celery.result.TaskSetResult (app.TaskSetResult)
- **{app.AsyncResult}**
    - celery.result.BaseAsyncResult / celery.result.AsyncResult
- **celery.bin.celeryd.WorkerCommand**
    - **celery.apps.worker.Worker**
        * **celery.worker.WorkerController**
            · **celery.worker.consumer.Consumer**

              celery.worker.job.TaskRequest

              celery.events.EventDispatcher

              celery.worker.control.ControlDispatch

                celery.woker.control.registry.Panel

                celery.pidbox.BroadcastPublisher

              celery.pidbox.BroadcastConsumer

            · celery.worker.controllers.Mediator

            · celery.beat.EmbeddedService
- **celery.bin.celeryev.run_celeryev**
    - **celery.events.snapshot.evcam**
        * celery.events.snapshot.Polaroid
        * celery.events.EventReceiver
    - **celery.events.cursesmon.evtop**
        * celery.events.EventReceiver
        * celery.events.cursesmon.CursesMonitor
    - **celery.events.dumper**
        * celery.events.EventReceiver
- **celery.bin.celeryctl.celeryctl**
    - celery.bin.celeryctl.Command
- celery.bin.caqmadm.AMQPAdmin
- **celery.bin.celerybeat.BeatCommand**
    - **celery.apps.beat.Beat**
        * **celery.beat.Service**
            · celery.beat.Scheduler

11.5 Internal Module Reference

Release 2.4

Date February 04, 2014

11.5.1 celery.worker

- celery.worker

celery.worker

The worker.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

`celery.worker.`**`WORKER_SIGIGNORE = frozenset(['SIGINT'])`**
List of signals to ignore when a child process starts.

`celery.worker.`**`WORKER_SIGRESET = frozenset(['SIGHUP', 'SIGTERM', 'SIGTTOU', 'SIGTTIN', 'SIGUSR1'])`**
List of signals to reset when a child process starts.

class `celery.worker.`**`WorkController`**(*concurrency=None*, *logfile=None*, *loglevel=None*, *send_events=None*, *hostname=None*, *ready_callback=<function noop at 0x2ef6848>*, *embed_clockservice=False*, *pool_cls=None*, *consumer_cls=None*, *mediator_cls=None*, *eta_scheduler_cls=None*, *schedule_filename=None*, *task_time_limit=None*, *task_soft_time_limit=None*, *max_tasks_per_child=None*, *pool_putlocks=None*, *db=None*, *prefetch_multiplier=None*, *eta_scheduler_precision=None*, *disable_rate_limits=None*, *autoscale=None*, *autoscaler_cls=None*, *scheduler_cls=None*, *app=None*)

Unmanaged worker instance.

`CLOSE = 2`

`RUN = 1`

`TERMINATE = 3`

`concurrency = None`
The number of simultaneous processes doing work (default: `CELERYD_CONCURRENCY`)

`consumer = None`
Consumer instance.

`embed_clockservice = None`
If `True`, celerybeat is embedded, running in the main worker process as a thread.

`logfile = None`
The logfile used, if no logfile is specified it uses *stderr* (default: `CELERYD_LOG_FILE`).

`logger = None`
The `logging.Logger` instance used for logging.

`loglevel = 40`
The loglevel used (default: `logging.INFO`)

`mediator = None`
Instance of `celery.worker.mediator.Mediator`.

on_timer_error(*exc_info*)

on_timer_tick(*delay*)

pool = None
The pool instance used.

process_task(*request*)
Process task by sending it to the pool of workers.

ready_queue = None
The internal queue object that holds tasks ready for immediate processing.

send_events = False
Enable the sending of monitoring events, these events can be captured by monitors (celerymon).

start()
Starts the workers main loop.

stop(*in_sighandler=False*)
Graceful shutdown of the worker server.

terminate(*in_sighandler=False*)
Not so graceful shutdown of the worker server.

celery.worker.**process_initializer**(*app*, *hostname*)
Initializes the process so it can be used to process tasks.

Used for multiprocessing environments.

11.5.2 celery.worker.consumer

- celery.worker.consumer

celery.worker.consumer

This module contains the component responsible for consuming messages from the broker, processing the messages and keeping the broker connections up and running.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

- `start()` is an infinite loop, which only iterates again if the connection is lost. For each iteration (at start, or if the connection is lost) it calls `reset_connection()`, and starts the consumer by calling `consume_messages()`.
- `reset_connection()`, clears the internal queues, establishes a new connection to the broker, sets up the task consumer (+ QoS), and the broadcast remote control command consumer.

  Also if events are enabled it configures the event dispatcher and starts up the heartbeat thread.
- Finally it can consume messages. `consume_messages()` is simply an infinite loop waiting for events on the AMQP channels.

  Both the task consumer and the broadcast consumer uses the same callback: `receive_message()`.

- So for each message received the `receive_message()` method is called, this checks the payload of the message for either a *task* key or a *control* key.

  If the message is a task, it verifies the validity of the message converts it to a `celery.worker.job.TaskRequest`, and sends it to `on_task()`.

  If the message is a control command the message is passed to `on_control()`, which in turn dispatches the control command using the control dispatcher.

  It also tries to handle malformed or invalid messages properly, so the worker doesn't choke on them and die. Any invalid messages are acknowledged immediately and logged, so the message is not resent again, and again.

- If the task has an ETA/countdown, the task is moved to the *eta_schedule* so the `timer2.Timer` can schedule it at its deadline. Tasks without an eta are moved immediately to the *ready_queue*, so they can be picked up by the `Mediator` to be sent to the pool.

- When a task with an ETA is received the QoS prefetch count is also incremented, so another message can be reserved. When the ETA is met the prefetch count is decremented again, though this cannot happen immediately because amqplib doesn't support doing broker requests across threads. Instead the current prefetch count is kept as a shared counter, so as soon as `consume_messages()` detects that the value has changed it will send out the actual QoS event to the broker.

- Notice that when the connection is lost all internal queues are cleared because we can no longer ack the messages reserved in memory. However, this is not dangerous as the broker will resend them to another worker when the channel is closed.

- **WARNING**: `stop()` does not close the connection! This is because some pre-acked messages may be in processing, and they need to be finished before the channel is closed. For celeryd this means the pool must finish the tasks it has acked early, *then* close the connection.

class `celery.worker.consumer.`**Consumer**(*ready_queue*, *eta_schedule*, *logger*, *init_callback=<function noop at 0x2ef6848>*, *send_events=False*, *hostname=None*, *initial_prefetch_count=2*, *pool=None*, *app=None*, *priority_timer=None*, *controller=None*)

Listen for messages received from the broker and move them to the ready queue for task processing.

Parameters

- **ready_queue** – See `ready_queue`.
- **eta_schedule** – See `eta_schedule`.

apply_eta_task(*task*)

Method called by the timer to apply a task with an ETA/countdown.

broadcast_consumer = None

The consumer used to consume broadcast commands.

close_connection()

Closes the current broker connection and all open channels.

connection = None

The broker connection.

consume_messages()

Consume messages forever (or until an exception is raised).

eta_schedule = None

Timer for tasks with an ETA/countdown.

event_dispatcher = None

A `celery.events.EventDispatcher` for sending events.

heart = None
The thread that sends event heartbeats at regular intervals. The heartbeats are used by monitors to detect that a worker went offline/disappeared.

hostname = None
The current hostname. Defaults to the system hostname.

info
Returns information about this consumer instance as a dict.

This is also the consumer related info returned by `celeryctl stats`.

init_callback = None
Optional callback to be called when the connection is established. Will only be called once, even if the connection is lost and re-established.

initial_prefetch_count = 0
Initial QoS prefetch count for the task channel.

logger = None
The logger instance to use. Defaults to the default Celery logger.

maybe_conn_error(*fun*)
Applies function but ignores any connection or channel errors raised.

on_control(*body*, *message*)
Process remote control command message.

on_decode_error(*message*, *exc*)
Callback called if an error occurs while decoding a message received.

Simply logs the error and acknowledges the message so it doesn't enter a loop.

Parameters

- **message** – The message with errors.
- **exc** – The original exception instance.

on_task(*task*)
Handle received task.

If the task has an *eta* we enter it into the ETA schedule, otherwise we move it the ready queue for immediate processing.

pidbox_node = None
The process mailbox (kombu pidbox node).

pool = None
The current worker pool instance.

priority_timer = None
A timer used for high-priority internal tasks, such as sending heartbeats.

ready_queue = None
The queue that holds tasks ready for immediate processing.

receive_message(*body*, *message*)
Handles incoming messages.

Parameters

- **body** – The message body.
- **message** – The kombu message object.

reset_connection()
Re-establish the broker connection and set up consumers, heartbeat and the event dispatcher.

reset_pidbox_node()
Sets up the process mailbox.

restart_heartbeat()
Restart the heartbeat thread.

This thread sends heartbeat events at intervals so monitors can tell if the worker is off-line/missing.

send_events = False
Enable/disable events.

start()
Start the consumer.

Automatically survives intermittent connection failure, and will retry establishing the connection and restart consuming messages.

stop()
Stop consuming.

Does not close the broker connection, so be sure to call `close_connection()` when you are finished with it.

stop_consumers(*close_connection=True*)
Stop consuming tasks and broadcast commands, also stops the heartbeat thread and event dispatcher.
Parameters **close_connection** – Set to False to skip closing the broker connection.

stop_pidbox_node()

task_consumer = None
The consumer used to consume task messages.

celery.worker.consumer.**INVALID_TASK_ERROR = 'Received invalid task message: %s\nThe message has been ignored**
Error message for when an invalid task message is received.

celery.worker.consumer.**PREFETCH_COUNT_MAX = 65535**
Prefetch count can't exceed short.

class celery.worker.consumer.**QoS**(*consumer*, *initial_value*, *logger*)
Quality of Service for Channel.

For thread-safe increment/decrement of a channels prefetch count value.
Parameters
- **consumer** – A `kombu.messaging.Consumer` instance.
- **initial_value** – Initial prefetch count value.
- **logger** – Logger used to log debug messages.

decrement(*n=1*)
Decrement the current prefetch count value by n.

decrement_eventually(*n=1*)
Decrement the value, but do not update the qos.

The MainThread will be responsible for calling `update()` when necessary.

increment(*n=1*)
Increment the current prefetch count value by n.

prev = None

set(*pcount*)
Set channel prefetch_count setting.

update()
Update prefetch count with current value.

celery.worker.consumer.**UNKNOWN_TASK_ERROR = 'Received unregistered task of type %s.\nThe message has been ign**
Error message for when an unregistered task is received.

11.5.3 celery.worker.job

- celery.worker.job

celery.worker.job

This module defines the `TaskRequest` class, which specifies how tasks are executed.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

exception celery.worker.job.**InvalidTaskError**
The task has invalid data or is not properly constructed.

class celery.worker.job.**TaskRequest**(*task_name*, *task_id*, *args*, *kwargs*, *on_ack=<function noop at 0x2ef6848>*, *retries=0*, *delivery_info=None*, *hostname=None*, *logger=None*, *eventer=None*, *eta=None*, *expires=None*, *app=None*, *taskset_id=None*, *chord=None*, ***opts*)
A request for task execution.

acknowledge()
Acknowledge task.

acknowledged = False
Flag set when the task has been acknowledged.

args = None
List of positional arguments to apply to the task.

chord = None
Body of a chord depending on this task.

delivery_info = None
Additional delivery info, e.g. contains the path from Producer to consumer.

error_msg = ' Task %(name)s[%(id)s] raised exception: %(exc)s\n%(traceback)s\n '
Format string used to log task failure.

eta = None
The tasks eta (for information only).

execute(*loglevel=None*, *logfile=None*)
Execute the task in a `WorkerTaskTrace`.
Parameters
- **loglevel** – The loglevel used by the task.
- **logfile** – The logfile used by the task.

execute_using_pool(*pool*, *loglevel=None*, *logfile=None*)
Like `execute()`, but using the `multiprocessing` pool.
Parameters

- **pool** – A `multiprocessing.Pool` instance.
- **loglevel** – The loglevel used by the task.
- **logfile** – The logfile used by the task.

expires = None
When the task expires.

extend_with_default_kwargs(*loglevel*, *logfile*)
Extend the tasks keyword arguments with standard task arguments.

Currently these are *logfile*, *loglevel*, *task_id*, *task_name*, *task_retries*, and *delivery_info*.

See `celery.task.base.Task.run()` for more information.

Magic keyword arguments are deprecated and will be removed in version 3.0.

classmethod from_message(*message*, *body*, *on_ack=<function noop at 0x2ef6848>*, ***kw*)
Create request from a task message.

Raises UnknownTaskError if the message does not describe a task, the message is also rejected.

get_instance_attrs(*loglevel*, *logfile*)

info(*safe=False*)

kwargs = None
Mapping of keyword arguments to apply to the task.

maybe_expire()
If expired, mark the task as revoked.

message = None
The message object. Used to acknowledge the message.

name = None
Kind of task. Must be a name registered in the task registry.

on_accepted(*pid*, *time_accepted*)
Handler called when task is accepted by worker pool.

on_ack = None
Callback called when the task should be acknowledged.

on_failure(*exc_info*)
Handler called if the task raised an exception.

on_retry(*exc_info*)
Handler called if the task should be retried.

on_success(*ret_value*)
Handler called if the task was successfully processed.

on_timeout(*soft*, *timeout*)
Handler called if the task times out.

repr_result(*result*, *maxlen=46*)

retries = 0
Number of times the task has been retried.

retry_msg = 'Task %(name)s[%(id)s] retry: %(exc)s'
Format string used to log task retry.

revoked()
If revoked, skip task and mark state.

send_event(*type*, ***fields*)

shortinfo()

success_msg = ' Task %(name)s[%(id)s] succeeded in %(runtime)ss: %(return_value)s\n '
Format string used to log task success.

task = None
The task class (set by constructor using `task_name`).

task_id = None
UUID of the task.

taskset_id = None
UUID of the taskset that this task belongs to.

terminate(*pool*, *signal=None*)

time_start = None
Timestamp set when the task is started.

worker_pid = None
Process id of the worker processing this task (if any).

`celery.worker.job.`**WANTED_DELIVERY_INFO = ('exchange', 'routing_key', 'consumer_tag')**
Keys to keep from the message delivery info. The values of these keys must be pickleable.

class `celery.worker.job.`**WorkerTaskTrace**(**args*, ***kwargs*)
Wraps the task in a jail, catches all exceptions, and saves the status and result of the task execution to the task meta backend.

If the call was successful, it saves the result to the task result backend, and sets the task status to *"SUCCESS"*.

If the call raises `RetryTaskError`, it extracts the original exception, uses that as the result and sets the task status to *"RETRY"*.

If the call results in an exception, it saves the exception as the task result, and sets the task status to *"FAILURE"*.

Parameters

- **task_name** – The name of the task to execute.
- **task_id** – The unique id of the task.
- **args** – List of positional args to pass on to the function.
- **kwargs** – Keyword arguments mapping to pass on to the function.
- **loader** – Custom loader to use, if not specified the current app loader will be used.
- **hostname** – Custom hostname to use, if not specified the system hostname will be used.

Returns the evaluated functions return value on success, or the exception instance on failure.

execute()
Execute, trace and store the result of the task.

execute_safe(**args*, ***kwargs*)
Same as `execute()`, but catches errors.

handle_failure(*exc*, *type_*, *tb*, *strtb*)
Handle exception.

handle_retry(*exc*, *type_*, *tb*, *strtb*)
Handle retry exception.

handle_success(*retval*, **args*)
Handle successful execution.

hostname = None
Hostname to report as.

loader = None
Current loader.

celery.worker.job.**default_encode**(*obj*)

celery.worker.job.**execute_and_trace**(*task_name*, **args*, ***kwargs*)
This is a pickleable method used as a target when applying to pools.

It's the same as:

```
>>> WorkerTaskTrace(task_name, *args, **kwargs).execute_safe()
```

11.5.4 celery.worker.mediator

- celery.worker.mediator

celery.worker.mediator

The mediator is an internal thread that moves tasks from an internal `Queue` to the worker pool.

This is only used if rate limits are enabled, as it moves messages from the rate limited queue (which holds tasks that are allowed to be processed) to the pool. Disabling rate limits will also disable this machinery, and can improve performance.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.worker.mediator.**Mediator**(*ready_queue*, *callback*, *logger=None*, *app=None*)

callback = None
Callback called when a task is obtained.

move()

ready_queue = None
The task queue, a `Queue` instance.

run()
Move tasks until `stop()` is called.

stop()
Gracefully shutdown the thread.

11.5.5 celery.worker.buckets

- celery.worker.buckets

celery.worker.buckets

This module implements the rate limiting of tasks, by having a token bucket queue for each task type. When a task is allowed to be processed it's moved over the the `ready_queue`

The `celery.worker.mediator` is then responsible for moving tasks from the `ready_queue` to the worker pool.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.worker.buckets.`**FastQueue**(*maxsize=0*)
`Queue.Queue` supporting the interface of `TokenBucketQueue`.

clear()

expected_time(*tokens=1*)

items

wait(*block=True*)

exception `celery.worker.buckets.`**RateLimitExceeded**
The token buckets rate limit has been exceeded.

class `celery.worker.buckets.`**TaskBucket**(*task_registry*)
This is a collection of token buckets, each task type having its own token bucket. If the task type doesn't have a rate limit, it will have a plain `Queue` object instead of a `TokenBucketQueue`.

The `put()` operation forwards the task to its appropriate bucket, while the `get()` operation iterates over the buckets and retrieves the first available item.

Say we have three types of tasks in the registry: *celery.ping*, *feed.refresh* and *video.compress*, the TaskBucket will consist of the following items:

```
{"celery.ping": TokenBucketQueue(fill_rate=300),
 "feed.refresh": Queue(),
 "video.compress": TokenBucketQueue(fill_rate=2)}
```

The get operation will iterate over these until one of the buckets is able to return an item. The underlying datastructure is a *dict*, so the order is ignored here.

Parameters **task_registry** – The task registry used to get the task type class for a given task name.

add_bucket_for_type(*task_name*)
Add a bucket for a task type.

Will read the tasks rate limit and create a `TokenBucketQueue` if it has one. If the task doesn't have a rate limit `FastQueue` will be used instead.

clear()
Delete the data in all of the buckets.

empty()
Returns `True` if all of the buckets are empty.

get(*block=True*, *timeout=None*)
Retrive the task from the first available bucket.

Available as in, there is an item in the queue and you can consume tokens from it.

get_bucket_for_type(*task_name*)
Get the bucket for a particular task type.

get_nowait()

init_with_registry()
Initialize with buckets for all the task types in the registry.

items
Flattens the data in all of the buckets into a single list.

put(*request*)
Put a TaskRequest into the appropiate bucket.

put_nowait(*request*)
Put a TaskRequest into the appropiate bucket.

qsize()
Get the total size of all the queues.

refresh()
Refresh rate limits for all task types in the registry.

update_bucket_for_type(*task_name*)

class celery.worker.buckets.**TokenBucketQueue**(*fill_rate*, *queue=None*, *capacity=1*)
Queue with rate limited get operations.

This uses the token bucket algorithm to rate limit the queue on get operations.

Parameters

- **fill_rate** – The rate in tokens/second that the bucket will be refilled.
- **capacity** – Maximum number of tokens in the bucket. Default is 1.

exception RateLimitExceeded
The token buckets rate limit has been exceeded.

TokenBucketQueue.**clear**()
Delete all data in the queue.

TokenBucketQueue.**empty**()
Returns True if the queue is empty.

TokenBucketQueue.**expected_time**(*tokens=1*)
Returns the expected time in seconds of when a new token should be available.

TokenBucketQueue.**get**(*block=True*)
Remove and return an item from the queue.

Raises

- **RateLimitExceeded** – If a token could not be consumed from the token bucket (consuming from the queue too fast).
- **Queue.Empty** – If an item is not immediately available.

TokenBucketQueue.**get_nowait**()
Remove and return an item from the queue without blocking.

Raises

- **RateLimitExceeded** – If a token could not be consumed from the token bucket (consuming from the queue too fast).
- **Queue.Empty** – If an item is not immediately available.

TokenBucketQueue.**items**
Underlying data. Do not modify.

TokenBucketQueue.**put**(*item*, *block=True*)
Put an item onto the queue.

TokenBucketQueue.**put_nowait**(*item*)
Put an item into the queue without blocking.
Raises Queue.Full If a free slot is not immediately available.

TokenBucketQueue.**qsize**()
Returns the size of the queue.

TokenBucketQueue.**wait**(*block=False*)
Wait until a token can be retrieved from the bucket and return the next item.

celery.worker.buckets.**chain_from_iterable**()
chain.from_iterable(iterable) –> chain object

Alternate chain() contructor taking a single iterable argument that evaluates lazily.

11.5.6 celery.worker.heartbeat

- celery.worker.heartbeat

celery.worker.heartbeat

This is the internal thread that sends heartbeat events at regular intervals.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.worker.heartbeat.**Heart**(*timer*, *eventer*, *interval=None*)
Timer sending heartbeats at regular intervals.
Parameters
- **timer** – Timer instance.
- **eventer** – Event dispatcher used to send the event.
- **interval** – Time in seconds between heartbeats. Default is 30 seconds.

start()

stop()

11.5.7 celery.worker.state

- celery.worker.state

celery.worker.state

Internal worker state (global)

This includes the currently active and reserved tasks, statistics, and revoked tasks.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.worker.state.**Persistent**(*filename*)

close()

db

merge(*d*)

open()

save()

storage = <module 'shelve' from '/usr/lib/python2.7/shelve.pyc'>

sync(*d*)

celery.worker.state.**REVOKES_MAX = 10000**
maximum number of revokes to keep in memory.

celery.worker.state.**REVOKE_EXPIRES = 3600**
how many seconds a revoke will be active before being expired when the max limit has been exceeded.

celery.worker.state.**SOFTWARE_INFO = {'sw_sys': 'Linux', 'sw_ident': 'celeryd', 'sw_ver': '2.4.7'}**
Worker software/platform information.

celery.worker.state.**active_requests = set([])**
set of currently active `TaskRequest`'s.

celery.worker.state.**reserved_requests = set([])**
set of all reserved `TaskRequest`'s.

celery.worker.state.**revoked = LimitedSet([[]])**
the list of currently revoked tasks. Persistent if statedb set.

celery.worker.state.**task_accepted**(*request*)
Updates global state when a task has been accepted.

celery.worker.state.**task_ready**(*request*)
Updates global state when a task is ready.

celery.worker.state.**task_reserved**(*request*)
Updates global state when a task has been reserved.

celery.worker.state.**total_count = defaultdict(<function <lambda> at 0x729c938>, {})**
count of tasks executed by the worker, sorted by type.

11.5.8 celery.worker.autoscale

- celery.worker.autoscale

celery.worker.autoscale

This module implements the internal thread responsible for growing and shrinking the pool according to the current autoscale settings.

The autoscale thread is only enabled if autoscale has been enabled on the command line.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.worker.autoscale.**Autoscaler**(*pool*, *max_concurrency*, *min_concurrency=0*, *keepalive=30*, *logger=None*)

force_scale_down(*n*)

force_scale_up(*n*)

info()

processes

qty

run()

scale()

scale_down(*n*)

scale_up(*n*)

stop()

update(*max=None*, *min=None*)

11.5.9 celery.concurrency

celery.concurrency.**get_implementation**(*cls*)

11.5.10 celery.concurrency.solo

class celery.concurrency.solo.**TaskPool**(*limit=None*, *putlocks=True*, *logger=None*, ***options*)
Solo task pool (blocking, inline).

on_apply(*target*, *args*, *kwargs*, *callback=None*, *accept_callback=None*, ***_*)

on_start()

11.5.11 celery.concurrency.processes

class celery.concurrency.processes.**TaskPool**(*limit=None*, *putlocks=True*, *logger=None*, ***options*)
Process Pool for processing tasks in parallel.

Parameters

- **processes** – see processes.
- **logger** – see logger.

limit
The number of processes that can run simultaneously.

logger
The logger used for debugging.

class **Pool**(*processes=None*, *initializer=None*, *initargs=()*, *maxtasksperchild=None*, *timeout=None*, *soft_timeout=None*)
Class which supports an async version of the *apply()* builtin

class **Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*)
Process objects represent activity that is run in a separate process

The class is analagous to *threading.Thread*

authkey

daemon
Return whether process is a daemon

exitcode
Return exit code of process or *None* if it has yet to stop

ident
Return identifier (PID) of process or *None* if it has yet to start

is_alive()
Return whether process is alive

join(*timeout=None*)
Wait until child process terminates

name

pid
Return identifier (PID) of process or *None* if it has yet to start

run()
Method to be run in sub-process; can be overridden in sub-class

start()
Start child process

terminate()
Terminate process; sends SIGTERM signal or uses TerminateProcess()

class TaskPool.Pool.**ResultHandler**(*outqueue*, *get*, *cache*, *poll*, *join_exited_workers*, *putlock*)

body()

exception TaskPool.Pool.**SoftTimeLimitExceeded**
The soft time limit has been exceeded. This exception is raised to give the task a chance to clean up.

class TaskPool.Pool.**Supervisor**(*pool*)

body()

class TaskPool.Pool.**TaskHandler**(*taskqueue*, *put*, *outqueue*, *pool*)

body()

class TaskPool.Pool.**TimeoutHandler**(*processes*, *cache*, *t_soft*, *t_hard*)

body()

TaskPool.Pool.**apply**(*func*, *args=()*, *kwds={}*)
Equivalent of *apply()* builtin

TaskPool.Pool.**apply_async**(*func*, *args=()*, *kwds={}*, *callback=None*, *accept_callback=None*, *timeout_callback=None*, *waitforslot=False*, *error_callback=None*, *soft_timeout=None*, *timeout=None*)
Asynchronous equivalent of *apply()* builtin.

Callback is called when the functions return value is ready. The accept callback is called when the job is accepted to be executed.

Simplified the flow is like this:

```
>>> if accept_callback:
...     accept_callback()
>>> retval = func(*args, **kwds)
>>> if callback:
...     callback(retval)
```

TaskPool.Pool.**close**()

TaskPool.Pool.**grow**(*n=1*)

TaskPool.Pool.**imap**(*func*, *iterable*, *chunksize=1*)
Equivalent of *itertools.imap()* – can be MUCH slower than *Pool.map()*

TaskPool.Pool.**imap_unordered**(*func*, *iterable*, *chunksize=1*)
Like *imap()* method but ordering of results is arbitrary

TaskPool.Pool.**join**()

TaskPool.Pool.**map**(*func*, *iterable*, *chunksize=None*)
Equivalent of *map()* builtin

TaskPool.Pool.**map_async**(*func*, *iterable*, *chunksize=None*, *callback=None*)
Asynchronous equivalent of *map()* builtin

TaskPool.Pool.**shrink**(*n=1*)

TaskPool.Pool.**terminate**()

TaskPool.**grow**(*n=1*)

TaskPool.**num_processes**

TaskPool.**on_start**()
Run the task pool.

Will pre-fork all workers so they're ready to accept tasks.

TaskPool.**on_stop**()
Gracefully stop the pool.

TaskPool.**on_terminate**()
Force terminate the pool.

TaskPool.**shrink**(*n=1*)

TaskPool.**terminate_job**(*pid*, *signal=None*)

11.5.12 celery.concurrency.processes.pool

class celery.concurrency.processes.pool.**ApplyResult**(*cache*, *callback*, *accept_callback=None*, *timeout_callback=None*, *error_callback=None*, *soft_timeout=None*, *timeout=None*)

accepted()

get(*timeout=None*)

ready()

successful()

wait(*timeout=None*)

worker_pids()

celery.concurrency.processes.pool.**DynamicPool**
alias of Pool

class celery.concurrency.processes.pool.**IMapIterator**(*cache*)

next(*timeout=None*)

class celery.concurrency.processes.pool.**IMapUnorderedIterator**(*cache*)

class celery.concurrency.processes.pool.**LaxBoundedSemaphore**(*value=1*, *verbose=None*)
Semaphore that checks that # release is <= # acquires, but ignores if # releases >= value.

clear()

release()

class celery.concurrency.processes.pool.**MapResult**(*cache*, *chunksize*, *length*, *callback*)

accepted()

worker_pids()

exception celery.concurrency.processes.pool.**MaybeEncodingError**(*exc*, *value*)
Wraps unpickleable object.

class celery.concurrency.processes.pool.**Pool**(*processes=None*, *initializer=None*, *initargs=()*, *maxtasksperchild=None*, *timeout=None*, *soft_timeout=None*)
Class which supports an async version of the *apply()* builtin

class Process(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*)
Process objects represent activity that is run in a separate process

The class is analagous to *threading.Thread*

authkey

daemon
Return whether process is a daemon

exitcode
Return exit code of process or *None* if it has yet to stop

ident
Return identifier (PID) of process or *None* if it has yet to start

is_alive()
Return whether process is alive

join(*timeout=None*)
Wait until child process terminates

name

pid
Return identifier (PID) of process or *None* if it has yet to start

run()
Method to be run in sub-process; can be overridden in sub-class

start()
Start child process

terminate()
Terminate process; sends SIGTERM signal or uses TerminateProcess()

class Pool.**ResultHandler**(*outqueue*, *get*, *cache*, *poll*, *join_exited_workers*, *putlock*)

body()

exception Pool.**SoftTimeLimitExceeded**
The soft time limit has been exceeded. This exception is raised to give the task a chance to clean up.

class Pool.**Supervisor**(*pool*)

body()

class Pool.**TaskHandler**(*taskqueue*, *put*, *outqueue*, *pool*)

body()

class Pool.**TimeoutHandler**(*processes*, *cache*, *t_soft*, *t_hard*)

body()

Pool.**apply**(*func*, *args=()*, *kwds={}*)
Equivalent of *apply()* builtin

Pool.**apply_async**(*func*, *args=()*, *kwds={}*, *callback=None*, *accept_callback=None*, *timeout_callback=None*, *waitforslot=False*, *error_callback=None*, *soft_timeout=None*, *timeout=None*)
Asynchronous equivalent of *apply()* builtin.

Callback is called when the functions return value is ready. The accept callback is called when the job is accepted to be executed.

Simplified the flow is like this:

```
>>> if accept_callback:
...     accept_callback()
>>> retval = func(*args, **kwds)
>>> if callback:
...     callback(retval)
```

Pool.**close**()

Pool.**grow**(*n=1*)

Pool.**imap**(*func*, *iterable*, *chunksize=1*)
Equivalent of *itertools.imap()* – can be MUCH slower than *Pool.map()*

Pool.**imap_unordered**(*func*, *iterable*, *chunksize=1*)
Like *imap()* method but ordering of results is arbitrary

Pool.**join**()

Pool.**map**(*func*, *iterable*, *chunksize=None*)
Equivalent of *map()* builtin

Pool.**map_async**(*func*, *iterable*, *chunksize=None*, *callback=None*)
Asynchronous equivalent of *map()* builtin

Pool.**shrink**(*n=1*)

Pool.**terminate**()

class celery.concurrency.processes.pool.**PoolThread**(**args*, ***kwargs*)

close()

run()

terminate()

class celery.concurrency.processes.pool.**ResultHandler**(*outqueue*, *get*, *cache*, *poll*, *join_exited_workers*, *putlock*)

body()

class celery.concurrency.processes.pool.**Supervisor**(*pool*)

body()

class celery.concurrency.processes.pool.**TaskHandler**(*taskqueue*, *put*, *outqueue*, *pool*)

body()

class celery.concurrency.processes.pool.**ThreadPool**(*processes=None*, *initializer=None*, *initargs=()*)

class DummyProcess(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*)

exitcode

start()

ThreadPool.**Process**
alias of DummyProcess

class celery.concurrency.processes.pool.**TimeoutHandler**(*processes*, *cache*, *t_soft*, *t_hard*)

body()

exception celery.concurrency.processes.pool.**WorkersJoined**
All workers have terminated.

celery.concurrency.processes.pool.**error**(*msg*, **args*, ***kwargs*)

celery.concurrency.processes.pool.**mapstar**(*args*)

celery.concurrency.processes.pool.**soft_timeout_sighandler**(*signum*, *frame*)

celery.concurrency.processes.pool.**worker**(*inqueue*, *outqueue*, *initializer=None*, *initargs=()*, *maxtasks=None*)

11.5.13 celery.concurrency.eventlet† (*experimental*)

class celery.concurrency.eventlet.**Schedule**(**args*, ***kwargs*)

clear()

enter(*entry*, *eta=None*, *priority=0*)

queue

class celery.concurrency.eventlet.**TaskPool**(**args*, ***kwargs*)

class Timer(*schedule=None*, *on_error=None*, *on_tick=None*, ***kwargs*)

class Schedule(**args*, ***kwargs*)

clear()

enter(*entry*, *eta=None*, *priority=0*)

queue

TaskPool.Timer.**cancel**(*tref*)

TaskPool.Timer.**ensure_started**()

TaskPool.Timer.**start**()

TaskPool.Timer.**stop**()

TaskPool.**is_green** = True

TaskPool.**on_apply**(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, ***_*)

TaskPool.**on_start**()

TaskPool.**on_stop**()

TaskPool.**signal_safe** = False

class celery.concurrency.eventlet.**Timer**(*schedule=None*, *on_error=None*, *on_tick=None*, ***kwargs*)

class Schedule(**args*, ***kwargs*)

clear()

enter(*entry*, *eta=None*, *priority=0*)

queue

Timer.**cancel**(*tref*)

Timer.**ensure_started**()

Timer.**start**()

Timer.**stop**()

celery.concurrency.eventlet.**apply_target**(*target*, *args=()*, *kwargs={}*, *callback=None*, *accept_callback=None*, *getpid=None*)

11.5.14 celery.concurrency.gevent† (*experimental*)

class celery.concurrency.gevent.**Schedule**(**args*, ***kwargs*)

clear()

enter(*entry*, *eta=None*, *priority=0*)

queue

class celery.concurrency.gevent.**TaskPool**(**args*, ***kwargs*)

class Timer(*schedule=None*, *on_error=None*, *on_tick=None*, ***kwargs*)

class Schedule(**args*, ***kwargs*)

clear()

enter(*entry*, *eta=None*, *priority=0*)

queue

TaskPool.Timer.**ensure_started**()

TaskPool.Timer.**start**()

TaskPool.Timer.**stop**()

TaskPool.**is_green** = True

TaskPool.**on_apply**(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, ***_*)

TaskPool.**on_start**()

TaskPool.**on_stop**()

TaskPool.**rlimit_safe** = False

TaskPool.**signal_safe** = False

class celery.concurrency.gevent.**Timer**(*schedule=None*, *on_error=None*, *on_tick=None*, ***kwargs*)

class Schedule(**args*, ***kwargs*)

clear()

enter(*entry*, *eta=None*, *priority=0*)

queue

Timer.**ensure_started**()

Timer.**start**()

Timer.**stop**()

11.5.15 celery.concurrency.base

class celery.concurrency.base.**BasePool**(*limit=None*, *putlocks=True*, *logger=None*, ***options*)

CLOSE = 2

RUN = 1

TERMINATE = 3

class Timer(*schedule=None*, *on_error=None*, *on_tick=None*, ***kwargs*)

class Entry(*fun*, *args=None*, *kwargs=None*)

cancel()

cancelled = False

class BasePool.Timer.**Schedule**(*max_interval=2*, *on_error=None*)
ETA scheduler.

clear()

empty()
Is the schedule empty?

enter(*entry*, *eta=None*, *priority=0*)
Enter function into the scheduler.

Parameters

- **entry** – Item to enter.
- **eta** – Scheduled time as a datetime.datetime object.
- **priority** – Unused.

handle_error(*exc_info*)

info()

on_error = None

queue

BasePool.Timer.**apply_after**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

BasePool.Timer.**apply_at**(*eta*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

BasePool.Timer.**apply_entry**(*entry*)

BasePool.Timer.**apply_interval**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

BasePool.Timer.**cancel**(*tref*)

BasePool.Timer.**clear**()

BasePool.Timer.**empty**()

BasePool.Timer.**ensure_started**()

BasePool.Timer.**enter**(*entry*, *eta*, *priority=None*)

BasePool.Timer.**enter_after**(*msecs*, *entry*, *priority=0*)

BasePool.Timer.**exit_after**(*msecs*, *priority=10*)

BasePool.Timer.**next**()

BasePool.Timer.**on_tick = None**

BasePool.Timer.**queue**

BasePool.Timer.**run**()

BasePool.Timer.**running = False**

BasePool.Timer.**stop**()

BasePool.**active**

BasePool.**apply_async**(*target*, *args=None*, *kwargs=None*, *callback=None*, *errback=None*, *accept_callback=None*, *timeout_callback=None*, *soft_timeout=None*, *timeout=None*, ***compat*)

Equivalent of the `apply()` built-in function.

Callbacks should optimally return as soon as possible ince otherwise the thread which handles the result will get blocked.

BasePool.**info**

BasePool.**is_green = False**

BasePool.**num_processes**

BasePool.**on_apply**(**args*, ***kwargs*)

BasePool.**on_ready**(*callback*, *errback*, *ret_value*)

What to do when a worker task is ready and its return value has been collected.

BasePool.**on_start**()

BasePool.**on_stop**()

BasePool.**on_terminate**()

BasePool.**on_worker_error**(*errback*, *exc_info*)

BasePool.**rlimit_safe = True**

BasePool.**safe_apply_callback**(*fun*, **args*)

BasePool.**signal_safe = True**

BasePool.**start**()

BasePool.**stop**()

BasePool.**terminate**()

BasePool.**terminate_job**(*pid*)

celery.concurrency.base.**apply_target**(*target*, *args=()*, *kwargs={}*, *callback=None*, *accept_callback=None*, *pid=None*)

11.5.16 celery.concurrency.threads‡ (minefield)

class `celery.concurrency.threads.`**`NullDict`**(*dict=None*, ***kwargs*)

class `celery.concurrency.threads.`**`TaskPool`**(**args*, ***kwargs*)

`on_apply`(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, ***_*)

`on_start`()

`on_stop`()

11.5.17 celery.beat

- celery.beat

celery.beat

The Celery periodic task scheduler.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

`celery.beat.`**`EmbeddedService`**(**args*, ***kwargs*)

Return embedded clock service.

Parameters **thread** – Run threaded instead of as a separate process. Default is `False`.

class `celery.beat.`**`PersistentScheduler`**(**args*, ***kwargs*)

`close`()

`get_schedule`()

`info`

`persistence` = <module 'shelve' from '/usr/lib/python2.7/shelve.pyc'>

`setup_schedule`()

`sync`()

class `celery.beat.`**`ScheduleEntry`**(*name=None*, *task=None*, *last_run_at=None*, *total_run_count=None*, *schedule=None*, *args=()*, *kwargs={}*, *options={}*, *relative=False*)

An entry in the scheduler.

Parameters

- **name** – see `name`.
- **schedule** – see `schedule`.
- **args** – see `args`.
- **kwargs** – see `kwargs`.
- **options** – see `options`.
- **last_run_at** – see `last_run_at`.
- **total_run_count** – see `total_run_count`.
- **relative** – Is the time relative to when the server starts?

args = None
Positional arguments to apply.

is_due()
See `celery.task.base.PeriodicTask.is_due()`.

kwargs = None
Keyword arguments to apply.

last_run_at = None
The time and date of when this task was last scheduled.

name = None
The task name

next(*last_run_at=None*)
Returns a new instance of the same class, but with its date and count fields updated.

options = None
Task execution options.

schedule = None
The schedule (run_every/crontab)

total_run_count = 0
Total number of times this task has been scheduled.

update(*other*)
Update values from another entry.

Does only update "editable" fields (task, schedule, args, kwargs, options).

class `celery.beat.`**Scheduler**(*schedule=None*, *logger=None*, *max_interval=None*, *app=None*, *Publisher=None*, *lazy=False*, ***kwargs*)
Scheduler for periodic tasks.

Parameters
- **schedule** – see `schedule`.
- **logger** – see `logger`.
- **max_interval** – see `max_interval`.

Entry
alias of `ScheduleEntry`

add(***kwargs*)

apply_async(*entry*, *publisher=None*, ***kwargs*)

close()

connection

get_schedule()

info

install_default_entries(*data*)

logger = None
Current logger.

max_interval = 1
Maximum time to sleep between re-checking the schedule.

maybe_due(*entry*, *publisher=None*)

merge_inplace(*b*)

publisher

reserve(*entry*)

schedule
The schedule dict/shelve.

send_task(**args*, ***kwargs*)

set_schedule(*schedule*)

setup_schedule()

should_sync()

sync()

sync_every = 180
How often to sync the schedule (3 minutes by default)

tick()
Run a tick, that is one iteration of the scheduler.

Executes all due tasks.

update_from_dict(*dict_*)

exception celery.beat.**SchedulingError**
An error occured while scheduling a task.

class celery.beat.**Service**(*logger=None*, *max_interval=None*, *schedule_filename=None*, *scheduler_cls=None*, *app=None*)

get_scheduler(*lazy=False*)

scheduler

scheduler_cls
alias of PersistentScheduler

start(*embedded_process=False*)

stop(*wait=False*)

sync()

11.5.18 celery.backends

celery.backends.**get_backend_cls**(**args*, ***kwargs*)
Get backend class by name/alias

11.5.19 celery.backends.base

celery.backends.base

class celery.backends.base.**BaseBackend**(**args*, ***kwargs*)
Base backend class.

EXCEPTION_STATES = frozenset([’FAILURE’, ‘RETRY’, ‘REVOKED’])

READY_STATES = frozenset([’FAILURE’, ‘REVOKED’, ‘SUCCESS’])

exception TimeoutError
The operation timed out.

BaseBackend.**UNREADY_STATES = frozenset(['STARTED', 'RECEIVED', 'RETRY', 'PENDING'])**

BaseBackend.**cleanup**()
Backend cleanup. Is run by celery.task.DeleteExpiredTaskMetaTask.

BaseBackend.**decode**(*payload*)

BaseBackend.**delete_taskset**(*taskset_id*)

BaseBackend.**encode**(*data*)

BaseBackend.**encode_result**(*result*, *status*)

BaseBackend.**exception_to_python**(*exc*)
Convert serialized exception to Python exception.

BaseBackend.**forget**(*task_id*)

BaseBackend.**get_result**(*task_id*)
Get the result of a task.

BaseBackend.**get_status**(*task_id*)
Get the status of a task.

BaseBackend.**get_traceback**(*task_id*)
Get the traceback for a failed task.

BaseBackend.**mark_as_done**(*task_id*, *result*)
Mark task as successfully executed.

BaseBackend.**mark_as_failure**(*task_id*, *exc*, *traceback=None*)
Mark task as executed with failure. Stores the execption.

BaseBackend.**mark_as_retry**(*task_id*, *exc*, *traceback=None*)
Mark task as being retries. Stores the current exception (if any).

BaseBackend.**mark_as_revoked**(*task_id*)

BaseBackend.**mark_as_started**(*task_id*, ***meta*)
Mark a task as started

BaseBackend.**on_chord_apply**(*setid*, *body*, **args*, ***kwargs*)

BaseBackend.**on_chord_part_return**(*task*)

BaseBackend.**prepare_exception**(*exc*)
Prepare exception for serialization.

BaseBackend.**prepare_expires**(*value*, *type=None*)

BaseBackend.**prepare_value**(*result*)
Prepare value for storage.

BaseBackend.**process_cleanup**()
Cleanup actions to do at the end of a task worker process.

BaseBackend.**reload_task_result**(*task_id*)
Reload task result, even if it has been previously fetched.

BaseBackend.**reload_taskset_result**(*task_id*)
Reload taskset result, even if it has been previously fetched.

BaseBackend.**restore_taskset**(*taskset_id*, *cache=True*)
Get the result of a taskset.

BaseBackend.**save_taskset**(*taskset_id*, *result*)
Store the result and status of a task.

BaseBackend.**store_result**(*task_id*, *result*, *status*, *traceback=None*)
Store the result and status of a task.

BaseBackend.**subpolling_interval = None**
Time to sleep between polling each individual item in *ResultSet.iterate*. as opposed to the *interval* argument which is for each pass.

BaseBackend.**wait_for**(*task_id*, *timeout=None*, *propagate=True*, *interval=0.5*)
Wait for task and return its result.

If the task raises an exception, this exception will be re-raised by `wait_for()`.

If *timeout* is not `None`, this raises the `celery.exceptions.TimeoutError` exception if the operation takes longer than *timeout* seconds.

class celery.backends.base.**BaseDictBackend**(**args*, ***kwargs*)

delete_taskset(*taskset_id*)

forget(*task_id*)

get_result(*task_id*)
Get the result of a task.

get_status(*task_id*)
Get the status of a task.

get_task_meta(*task_id*, *cache=True*)

get_taskset_meta(*taskset_id*, *cache=True*)

get_traceback(*task_id*)
Get the traceback for a failed task.

reload_task_result(*task_id*)

reload_taskset_result(*taskset_id*)

restore_taskset(*taskset_id*, *cache=True*)
Get the result for a taskset.

save_taskset(*taskset_id*, *result*)
Store the result of an executed taskset.

store_result(*task_id*, *result*, *status*, *traceback=None*, ***kwargs*)
Store task result and status.

class celery.backends.base.**DisabledBackend**(**args*, ***kwargs*)

get_result(**args*, ***kwargs*)

get_status(**args*, ***kwargs*)

get_traceback(**args*, ***kwargs*)

store_result(**args*, ***kwargs*)

wait_for(**args*, ***kwargs*)

class celery.backends.base.**KeyValueStoreBackend**(**args*, ***kwargs*)

delete(*key*)

get(*key*)

get_key_for_task(*task_id*)
Get the cache key for a task by id.

get_key_for_taskset(*taskset_id*)
Get the cache key for a task by id.

get_many(*task_ids*, *timeout=None*, *interval=0.5*)

mget(*keys*)

set(*key*, *value*)

task_keyprefix = 'celery-task-meta-'

taskset_keyprefix = 'celery-taskset-meta-'

celery.backends.base.**unpickle_backend**(*cls*, *args*, *kwargs*)
Returns an unpickled backend.

11.5.20 celery.backends.database

class celery.backends.database.**DatabaseBackend**(*dburi=None*, *expires=None*, *engine_options=None*, ***kwargs*)
The database result backend.

ResultSession()

cleanup()
Delete expired metadata.

subpolling_interval = 0.5

11.5.21 celery.backends.cache

class celery.backends.cache.**CacheBackend**(*expires=None*, *backend=None*, *options={}*, ***kwargs*)

client

delete(*key*)

get(*key*)

get_key_for_task(*task_id*)

get_key_for_taskset(*taskset_id*)

mget(*keys*)

servers = None

set(*key*, *value*)

class celery.backends.cache.**DummyClient**(**args*, ***kwargs*)

delete(*key*, **args*, ***kwargs*)

get(*key*, **args*, ***kwargs*)

get_multi(*keys*)

set(*key*, *value*, **args*, ***kwargs*)

celery.backends.cache.**get_best_memcache**(**args*, ***kwargs*)

celery.backends.cache.**import_best_memcache**()

11.5.22 celery.backends.amqp

class celery.backends.amqp.**AMQPBackend**(*connection=None*, *exchange=None*, *exchange_type=None*, *persistent=None*, *serializer=None*, *auto_delete=True*, ***kwargs*)

Publishes results by sending messages.

exception BacklogLimitExceeded

Too much state history to fast-forward.

class AMQPBackend.**Consumer**(*channel*, *queues=None*, *no_ack=None*, *auto_declare=None*, *callbacks=None*, *on_decode_error=None*)

Message consumer.

Parameters

- **channel** – see channel.
- **queues** – see queues.
- **no_ack** – see no_ack.
- **auto_declare** – see auto_declare
- **callbacks** – see callbacks.
- **on_decode_error** – see on_decode_error.

add_queue(*queue*)

auto_declare = True

callbacks = None

cancel()

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

cancel_by_queue(*queue*)

Cancel consumer by queue name.

channel = None

close()

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

connection

consume(*no_ack=None*)

consuming_from(*queue*)

declare()

Declare queues, exchanges and bindings.

This is done automatically at instantiation if auto_declare is set.

flow(*active*)
Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

no_ack = None

on_decode_error = None

purge()
Purge messages from all queues.

> **Warning:** This will *delete all ready messages*, there is no undo operation.

qos(*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the `no_ack` option is set.

Parameters

- **prefetch_size** – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply.
- **prefetch_count** – Specify the prefetch window in terms of whole messages.
- **apply_global** – Apply new settings globally on all channels. Currently not supported by RabbitMQ.

queues = None

receive(*body*, *message*)
Method called when a message is received.

This dispatches to the registered `callbacks`.

Parameters

- **body** – The decoded message body.
- **message** – The *Message* instance.

Raises **NotImplementedError** If no consumer callbacks have been registered.

recover(*requeue=False*)
Redeliver unacknowledged messages.

Asks the broker to redeliver all unacknowledged messages on the specified channel.

Parameters **requeue** – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

register_callback(*callback*)
Register a new callback to be called when a message is received.

The signature of the callback needs to accept two arguments: *(body, message)*, which is the decoded message body and the *Message* instance (a subclass of `Message`.

revive(*channel*)
Revive consumer after connection loss.

class `AMQPBackend.`**Exchange**(*name=''*, *type=''*, *channel=None*, ***kwargs*)
An Exchange declaration.

Parameters

- **name** – See `name`.
- **type** – See `type`.
- **channel** – See `channel`.
- **durable** – See `durable`.
- **auto_delete** – See `auto_delete`.
- **delivery_mode** – See `delivery_mode`.
- **arguments** – See `arguments`.

name
Name of the exchange. Default is no name (the default exchange).

type
AMQP defines four default exchange types (routing algorithms) that covers most of the common messaging use cases. An AMQP broker can also define additional exchange types, so see your broker manual for more information about available exchange types.

•*direct* (*default*)

Direct match between the routing key in the message, and the routing criteria used when a queue is bound to this exchange.

•*topic*

Wildcard match between the routing key and the routing pattern specified in the exchange/queue binding. The routing key is treated as zero or more words delimited by *"."* and supports special wildcard characters. *"*"* matches a single word and *"#"* matches zero or more words.

•*fanout*

Queues are bound to this exchange with no arguments. Hence any message sent to this exchange will be forwarded to all queues bound to this exchange.

•*headers*

Queues are bound to this exchange with a table of arguments containing headers and values (optional). A special argument named "x-match" determines the matching algorithm, where *"all"* implies an *AND* (all pairs must match) and *"any"* implies *OR* (at least one pair must match).

`arguments` is used to specify the arguments.

This description of AMQP exchange types was shamelessly stolen from the blog post AMQP in 10 minutes: Part 4 by Rajith Attapattu. This article is recommended reading.

channel
The channel the exchange is bound to (if bound).

durable

Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged when a server restarts. Default is `True`.

auto_delete
If set, the exchange is deleted when all queues have finished using it. Default is `False`.

delivery_mode
The default delivery mode used for messages. The value is an integer, or alias string.
- 1 or *"transient"*
  The message is transient. Which means it is stored in memory only, and is lost if the server dies or restarts.
- **2 or "persistent" (*default*)** The message is persistent. Which means the message is stored both in-memory, and on disk, and therefore preserved if the server dies or restarts.

The default value is 2 (persistent).

arguments
Additional arguments to specify when the exchange is declared.

Message(*body*, *delivery_mode=None*, *priority=None*, *content_type=None*, *content_encoding=None*, *properties=None*, *headers=None*)
Create message instance to be sent with `publish()`.

Parameters

- **body** – Message body.
- **delivery_mode** – Set custom delivery mode. Defaults to `delivery_mode`.
- **priority** – Message priority, 0 to 9. (currently not supported by RabbitMQ).
- **content_type** – The messages content_type. If content_type is set, no serialization occurs as it is assumed this is either a binary object, or you've done your own serialization. Leave blank if using built-in serialization as our library properly sets content_type.
- **content_encoding** – The character set in which this object is encoded. Use "binary" if sending in raw binary objects. Leave blank if using built-in serialization as our library properly sets content_encoding.
- **properties** – Message properties.
- **headers** – Message headers.

PERSISTENT_DELIVERY_MODE = 2

TRANSIENT_DELIVERY_MODE = 1

attrs = (('name', None), ('type', None), ('arguments', None), ('durable', <type 'bool'>), ('auto_delete', <type 'bool

auto_delete = False

can_cache_declaration

declare(*nowait=False*)
Declare the exchange.

Creates the exchange on the broker.

Parameters **nowait** – If set the server will not respond, and a response will not be waited for. Default is `False`.

delete(*if_unused=False*, *nowait=False*)
Delete the exchange declaration on server.

Parameters

- **if_unused** – Delete only if the exchange has no bindings. Default is `False`.
- **nowait** – If set the server will not respond, and a response will not be waited for. Default is `False`.

delivery_mode = 2

durable = True

name = ''

publish(*message*, *routing_key=None*, *mandatory=False*, *immediate=False*, *exchange=None*)
Publish message.

Parameters

- **message** – `Message()` instance to publish.
- **routing_key** – Routing key.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.

type = 'direct'

class AMQPBackend.**Producer**(*channel*, *exchange=None*, *routing_key=None*, *serializer=None*, *auto_declare=None*, *compression=None*, *on_return=None*)
Message Producer.

Parameters

- **channel** – Connection or channel.
- **exchange** – Optional default exchange.
- **routing_key** – Optional default routing key.
- **serializer** – Default serializer. Default is *"json"*.
- **compression** – Default compression method. Default is no compression.
- **auto_declare** – Automatically declare the default exchange at instantiation. Default is `True`.
- **on_return** – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to `publish()` is used. This callback needs the following signature: *(exception, exchange, routing_key, message)*. Note that the producer needs to drain events to use this feature.

auto_declare = True

channel = None

close()

compression = None

connection

declare()
Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

exchange = None

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
Declare the exchange if it hasn't already been declared during this session.

on_return = None

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=[]*, ***properties*)

Publish message to the specified exchange.

Parameters

- **body** – Message body.
- **routing_key** – Message routing key.
- **delivery_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content_type** – Content type. Default is auto-detect.
- **content_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** – Retry configuration, this is the keywords supported by `ensure()`.
- ****properties** – Additional message properties, see AMQP spec.

release()

revive(*channel*)

Revive the producer after connection loss.

routing_key = ''

serializer = None

class AMQPBackend.**Queue**(*name=''*, *exchange=None*, *routing_key=''*, *channel=None*, ***kwargs*)

A Queue declaration.

Parameters

- **name** – See `name`.
- **exchange** – See `exchange`.
- **routing_key** – See `routing_key`.
- **channel** – See `channel`.
- **durable** – See `durable`.
- **exclusive** – See `exclusive`.
- **auto_delete** – See `auto_delete`.
- **queue_arguments** – See `queue_arguments`.
- **binding_arguments** – See `binding_arguments`.

name
Name of the queue. Default is no name (default queue destination).

exchange
The `Exchange` the queue binds to.

routing_key
The routing key (if any), also called *binding key*.

The interpretation of the routing key depends on the `Exchange.type`.

- direct exchange
  Matches if the routing key property of the message and the `routing_key` attribute are identical.
- fanout exchange
  Always matches, even if the binding does not have a key.
- topic exchange
  Matches the routing key property of the message by a primitive pattern matching scheme. The message routing key then consists of words separated by dots (*"."*, like domain names), and two special characters are available; star (*"*"*) and hash (*"#"*). The star matches any word, and the hash matches zero or more words. For example *"*.stock.#"* matches the routing keys *"usd.stock"* and *"eur.stock.db"* but not *"stock.nasdaq"*.

channel
The channel the Queue is bound to (if bound).

durable
Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue.

Default is `True`.

exclusive
Exclusive queues may only be consumed from by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'.

Default is `False`.

auto_delete
If set, the queue is deleted when all consumers have finished using it. Last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted.

queue_arguments
Additional arguments used when declaring the queue.

binding_arguments
Additional arguments used when binding the queue.

alias
Unused in Kombu, but applications can take advantage of this. For example to give alternate names to queues with automatically generated queue names.

attrs = (('name', None), ('exchange', None), ('routing_key', None), ('queue_arguments', None), ('binding_argume

auto_delete = False

can_cache_declaration

cancel(*consumer_tag*)
Cancel a consumer by consumer tag.

consume(*consumer_tag=''*, *callback=None*, *no_ack=None*, *nowait=False*)
Start a queue consumer.

Consumers last as long as the channel they were created on, or until the client cancels them.

Parameters

- **consumer_tag** – Unique identifier for the consumer. The consumer tag is local to a connection, so two clients can use the same consumer tags. If this field is empty the server will generate a unique tag.
- **no_ack** – If set messages received does not have to be acknowledged.
- **nowait** – Do not wait for a reply.
- **callback** – callback called for each delivered message

declare(*nowait=False*)
Declares the queue, the exchange and binds the queue to the exchange.

delete(*if_unused=False*, *if_empty=False*, *nowait=False*)
Delete the queue.

Parameters

- **if_unused** – If set, the server will only delete the queue if it has no consumers. A channel error will be raised if the queue has consumers.
- **if_empty** – If set, the server will only delete the queue if it is empty. If it is not empty a channel error will be raised.
- **nowait** – Do not wait for a reply.

durable = True

exchange = <unbound Exchange (direct)>

exclusive = False

get(*no_ack=None*)
Poll the server for a new message.

Returns the message instance if a message was available, or `None` otherwise.

Parameters **no_ack** – If set messages received does not have to be acknowledged.

This method provides direct access to the messages in a queue using a synchronous dialogue, designed for specific types of applications where synchronous functionality is more important than performance.

name = ''

no_ack = False

purge(*nowait=False*)
Remove all ready messages from the queue.

queue_bind(*nowait=False*)
Create the queue binding on the server.

queue_declare(*nowait=False*, *passive=False*)
Declare queue on the server.

Parameters

- **nowait** – Do not wait for a reply.
- **passive** – If set, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state.

routing_key = ''

unbind()
Delete the binding on the server.

when_bound()

AMQPBackend.**consume**(*task_id*, *timeout=None*)

AMQPBackend.**delete_taskset**(*taskset_id*)

AMQPBackend.**drain_events**(*connection*, *consumer*, *timeout=None*, *now=<built-in function time>*)

AMQPBackend.**get_many**(*task_ids*, *timeout=None*, ***kwargs*)

AMQPBackend.**get_task_meta**(*task_id*, *cache=True*)

AMQPBackend.**poll**(*task_id*, *backlog_limit=100*)

AMQPBackend.**reload_task_result**(*task_id*)

AMQPBackend.**reload_taskset_result**(*task_id*)
Reload taskset result, even if it has been previously fetched.

AMQPBackend.**restore_taskset**(*taskset_id*, *cache=True*)

AMQPBackend.**revive**(*channel*)

AMQPBackend.**save_taskset**(*taskset_id*, *result*)

AMQPBackend.**wait_for**(*task_id*, *timeout=None*, *cache=True*, *propagate=True*, ***kwargs*)

exception celery.backends.amqp.**BacklogLimitExceeded**
Too much state history to fast-forward.

celery.backends.amqp.**repair_uuid**(*s*)

11.5.23 celery.backends.mongodb

MongoDB backend for celery.

class celery.backends.mongodb.**Bunch**(***kw*)

class celery.backends.mongodb.**MongoBackend**(**args*, ***kwargs*)

cleanup()
Delete expired metadata.

mongodb_database = 'celery'

mongodb_host = 'localhost'

mongodb_password = None

mongodb_port = 27017

mongodb_taskmeta_collection = 'celery_taskmeta'

mongodb_user = None

process_cleanup()

11.5.24 celery.backends.redis

class celery.backends.redis.**RedisBackend**(*host=None*, *port=None*, *db=None*, *password=None*, *expires=None*, ***kwargs*)

Redis task result store.

client

db = 0
default Redis db number (0)

delete(*key*)

get(*key*)

host = 'localhost'
default Redis server hostname (*localhost*).

mget(*keys*)

on_chord_apply(*setid*, **args*, ***kwargs*)

on_chord_part_return(*task*, *propagate=False*, *keyprefix='chord-unlock-%s'*)

password = None
default Redis password (None)

port = 6379
default Redis server port (6379)

redis = None
redis-py client module.

set(*key*, *value*)

11.5.25 celery.backends.cassandra

celery.backends.cassandra

class celery.backends.cassandra.**CassandraBackend**(*servers=None*, *keyspace=None*, *column_family=None*, *cassandra_options=None*, ***kwargs*)

Highly fault tolerant Cassandra backend.

servers
List of Cassandra servers with format: "hostname:port".

Raises celery.exceptions.ImproperlyConfigured if module pycassa is not available.

column_family = None

keyspace = None

process_cleanup()

servers = []

11.5.26 celery.backends.tyrant

celery.backends.tyrant

class celery.backends.tyrant.**TyrantBackend**(*tyrant_host=None*, *tyrant_port=None*, ***kwargs*)
Tokyo Cabinet based task backend store.

tyrant_host
The hostname to the Tokyo Tyrant server.

tyrant_port
The port to the Tokyo Tyrant server.

close()
Close the tyrant connection and remove the cache.

delete(*key*)

get(*key*)

open()
Get `pytyrant.PyTyrant` instance with the current server configuration.

The connection is then cached until you do an explicit `close()`.

process_cleanup()

set(*key*, *value*)

tyrant_host = **None**

tyrant_port = **None**

11.5.27 celery.execute.trace

- celery.execute.trace

celery.execute.trace

This module defines how the task execution is traced: errors are recorded, handlers are applied and so on.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.execute.trace.`**TaskTrace**(*task_name*, *task_id*, *args*, *kwargs*, *task=None*, *request=None*, *propagate=None*, ***_*)

execute()

handle_after_return(*status*, *retval*, *type_*, *tb*, *strtb*, *einfo=None*)

handle_failure(*exc*, *type_*, *tb*, *strtb*)
Handle exception.

handle_retry(*exc*, *type_*, *tb*, *strtb*)
Handle retry exception.

handle_success(*retval*, **args*)
Handle successful execution.

class `celery.execute.trace.`**TraceInfo**(*status='PENDING'*, *retval=None*, *exc_info=None*)

classmethod `trace` (*fun*, *args*, *kwargs*, *propagate=False*)
Trace the execution of a function, calling the appropiate callback if the function raises retry, an failure or returned successfully.
Parameters **propagate** – If true, errors will propagate to the caller.

11.5.28 celery.datastructures

Custom types and data structures.

copyright
3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

- TokenBucket
- AttributeDict
- DictAttribute
- ConfigurationView
- ExceptionInfo
- LimitedSet
- LRUCache

TokenBucket

class `celery.datastructures.`**`TokenBucket`** (*fill_rate*, *capacity=1*)
Token Bucket Algorithm.

See http://en.wikipedia.org/wiki/Token_Bucket Most of this code was stolen from an entry in the ASPN Python Cookbook: http://code.activestate.com/recipes/511490/

Thread safety

This implementation may not be thread safe.

`can_consume` (*tokens=1*)
Returns `True` if *tokens* number of tokens can be consumed from the bucket.

`capacity` = 1
Maximum number of tokensin the bucket.

`expected_time` (*tokens=1*)
Returns the expected time in seconds when a new token should be available.

Warning

This consumes a token from the bucket.

`fill_rate` = None
The rate in tokens/second that the bucket will be refilled

`timestamp` = None
Timestamp of the last time a token was taken out of the bucket.

AttributeDict

class celery.datastructures.**AttributeDict**
Dict subclass with attribute access.

class celery.datastructures.**AttributeDictMixin**
Adds attribute access to mappings.

d.key -> d[key]

DictAttribute

class celery.datastructures.**DictAttribute**(*obj*)
Dict interface to attributes.

obj[k] -> obj.k

get(*key*, *default=None*)

items()

iteritems()

setdefault(*key*, *default*)

ConfigurationView

class celery.datastructures.**ConfigurationView**(*changes*, *defaults*)
A view over an applications configuration dicts.

If the key does not exist in `changes`, the `defaults` dict is consulted.

Parameters

- **changes** – Dict containing changes to the configuration.
- **defaults** – Dict containing the default configuration.

changes = None

defaults = None

get(*key*, *default=None*)

items()

iteritems()

iterkeys()

itervalues()

keys()

setdefault(*key*, *default*)

update(**args*, ***kwargs*)

values()

ExceptionInfo

class celery.datastructures.**ExceptionInfo**(*exc_info*)
Exception wrapping an exception and its traceback.

Parameters **exc_info** – The exception info tuple as returned by `sys.exc_info()`.

exception = None
Exception instance.

tb = None
Pickleable traceback instance for use with `traceback`

traceback = None
String representation of the traceback.

type = None
Exception type.

LimitedSet

class `celery.datastructures.`**LimitedSet**(*maxlen=None*, *expires=None*)
Kind-of Set with limitations.

Good for when you need to test for membership (*a in set*), but the list might become to big, so you want to limit it so it doesn't consume too much resources.

Parameters

- **maxlen** – Maximum number of members before we start evicting expired members.
- **expires** – Time in seconds, before a membership expires.

add(*value*)
Add a new member.

as_dict()

chronologically

clear()
Remove all members

expires

first
Get the oldest member.

maxlen

pop_value(*value*)
Remove membership by finding value.

update(*other*)

LRUCache

class `celery.datastructures.`**LRUCache**(*limit=None*)
LRU Cache implementation using a doubly linked list to track access.

Parameters limit – The maximum number of keys to keep in the cache. When a new key is inserted and the limit has been exceeded, the *Least Recently Used* key will be discarded from the cache.

items()

iteritems()

itervalues()

keys()

values()

11.5.29 celery.routes

- celery.routes

celery.routes

Contains utilities for working with task routes (CELERY_ROUTES).

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.routes.**MapRoute**(*map*)
Creates a router out of a dict.

route_for_task(*task*, **args*, ***kwargs*)

class celery.routes.**Router**(*routes=None*, *queues=None*, *create_missing=False*, *app=None*)

expand_destination(*route*)

lookup_route(*task*, *args=None*, *kwargs=None*)

route(*options*, *task*, *args=()*, *kwargs={}*)

celery.routes.**prepare**(*routes*)
Expands the CELERY_ROUTES setting.

11.5.30 celery.log

class celery.log.**ColorFormatter**(*msg*, *use_color=True*)

COLORS = {'blue': <bound method colored.blue of ''>, 'black': <bound method colored.black of ''>, 'yellow': <bound me
Loglevel -> Color mapping.

colors = {'DEBUG': <bound method colored.blue of ''>, 'CRITICAL': <bound method colored.magenta of ''>, 'WARN

format(*record*)

formatException(*ei*)

class celery.log.**Logging**(*app*)

colored(*logfile=None*)

get_default_logger(*loglevel=None*, *name='celery'*)
Get default logger instance.
Parameters **loglevel** – Initial log level.

get_task_logger(*loglevel=None*, *name=None*)

redirect_stdouts_to_logger(*logger*, *loglevel=None*, *stdout=True*, *stderr=True*)
Redirect sys.stdout and sys.stderr to a logging instance.
Parameters
- **logger** – The logging.Logger instance to redirect to.

- **loglevel** – The loglevel redirected messages will be logged as.

setup_logger(*loglevel=None*, *logfile=None*, *format=None*, *colorize=None*, *name='celery'*, *root=True*, *app=None*, ***kwargs*)
Setup the `multiprocessing` logger.

If *logfile* is not specified, then *sys.stderr* is used.

Returns logger object.

setup_logging_subsystem(*loglevel=None*, *logfile=None*, *format=None*, *colorize=None*, ***kwargs*)

setup_task_logger(*loglevel=None*, *logfile=None*, *format=None*, *colorize=None*, *task_name=None*, *task_id=None*, *propagate=False*, *app=None*, ***kwargs*)
Setup the task logger.

If *logfile* is not specified, then *sys.stderr* is used.

Returns logger object.

supports_color(*logfile=None*)

class `celery.log.`**LoggingProxy**(*logger*, *loglevel=None*)
Forward file object to `logging.Logger` instance.

Parameters

- **logger** – The `logging.Logger` instance to forward to.
- **loglevel** – Loglevel to use when writing messages.

close()
When the object is closed, no write requests are forwarded to the logging object anymore.

closed = False

fileno()

flush()
This object is not buffered so any `flush()` requests are ignored.

isatty()
Always returns `False`. Just here for file support.

loglevel = 40

mode = 'w'

name = None

write(*data*)

writelines(*sequence*)
writelines(sequence_of_strings) -> None.

Write the strings to the file.

The sequence can be any iterable object producing strings. This is equivalent to calling `write()` for each string.

class `celery.log.`**SilenceRepeated**(*action*, *max_iterations=10*)
Only log action every n iterations.

11.5.31 celery.events.snapshot

- celery.events.snapshot

celery.events.snapshot

Consuming the events as a stream is not always suitable so this module implements a system to take snapshots of the state of a cluster at regular intervals. There is a full implementation of this writing the snapshots to a database in `djcelery.snapshots` in the *django-celery* distribution.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.events.snapshot.`**`Polaroid`**(*state*, *freq=1.0*, *maxrate=None*, *cleanup_freq=3600.0*, *logger=None*, *timer=None*, *app=None*)

`cancel`()

`capture`()

`cleanup`()

`cleanup_signal` = **<Signal: Signal>**

`clear_after` = **False**

`install`()

`on_cleanup`()

`on_shutter`(*state*)

`shutter`()

`shutter_signal` = **<Signal: Signal>**

`timer` = **<module 'celery.utils.timer2' from '/home/docs/checkouts/readthedocs.org/user_builds/celery/envs/2.4-archived/**

`celery.events.snapshot.`**`evcam`**(*camera*, *freq=1.0*, *maxrate=None*, *loglevel=0*, *logfile=None*, *pidfile=None*, *timer=None*, *app=None*)

11.5.32 celery.events.cursesmon

- celery.events.cursesmon

celery.events.cursesmon

Graphical monitor of Celery events using curses.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.events.cursesmon.**CursesMonitor**(*state*, *keymap=None*, *app=None*)

alert(*callback*, *title=None*)

alert_remote_control_reply(*reply*)

background = 7

display_height

display_task_row(*lineno*, *task*)

display_width

draw()

find_position()

foreground = 0

format_row(*uuid*, *task*, *worker*, *timestamp*, *state*)

greet = 'celeryev 2.4.7'

handle_keypress()

help = 'j:up k:down i:info t:traceback r:result c:revoke ^c: quit'

help_title = 'Keys: '

info_str = 'Info: '

init_screen()

keyalias = {258: 'J', 259: 'K', 343: 'I'}

keymap = {}

limit

move_selection(*direction=1*)

move_selection_down()

move_selection_up()

nap()

online_str = 'Workers online: '

readline(*x*, *y*)

resetscreen()

revoke_selection()

safe_add_str(*y*, *x*, *string*, **args*, ***kwargs*)

screen_delay = 10

screen_height

screen_width

selected_position = 0

selected_str = 'Selected: '

selected_task = None

selection_info()

selection_rate_limit()

selection_result()

selection_traceback()

tasks

win = None

workers

class celery.events.cursesmon.**DisplayThread**(*display*)

run()

celery.events.cursesmon.**evtop**(*app=None*)

11.5.33 celery.events.dumper

- celery.events.dumper

celery.events.dumper

THis is a simple program that dumps events to the console as they happen. Think of it like a *tcpdump* for Celery events.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.events.dumper.**Dumper**

format_task_event(*hostname*, *timestamp*, *type*, *task*, *event*)

on_event(*event*)

celery.events.dumper.**evdump**(*app=None*)

celery.events.dumper.**humanize_type**(*type*)

11.5.34 celery.db.models

class celery.db.models.**Task**(*task_id*)
Task result/status.

date_done

id

result

status

task_id

to_dict()

traceback

class celery.db.models.**TaskSet**(*taskset_id*, *result*)
TaskSet result

date_done

id

result

taskset_id

to_dict()

11.5.35 celery.db.session

celery.db.session.**ResultSession**(*dburi*, ***kwargs*)

celery.db.session.**create_session**(*dburi*, *short_lived_sessions=False*, ***kwargs*)

celery.db.session.**get_engine**(*dburi*, ***kwargs*)

celery.db.session.**setup_results**(*engine*)

11.5.36 celery.utils

- celery.utils

celery.utils

Utility functions.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

celery.utils.**abbr**(*S*, *max*, *ellipsis='...'*)

celery.utils.**abbrtask**(*S*, *max*)

celery.utils.**chunks**(*it*, *n*)
Split an iterator into chunks with *n* elements each.

Examples
n == 2 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 2) >>> list(x) [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10]]

n == 3 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 3) >>> list(x) [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]

celery.utils.**cry**()
Return stacktrace of all active threads.

From https://gist.github.com/737056

celery.utils.**cwd_in_path**(**args*, ***kwds*)

celery.utils.**deprecated**(*description=None*, *deprecation=None*, *removal=None*, *alternative=None*)

celery.utils.**find_module**(*module*, *path=None*, *imp=None*)
Version of `imp.find_module()` supporting dots.

celery.utils.**first**(*predicate*, *iterable*)
Returns the first element in *iterable* that *predicate* returns a `True` value for.

celery.utils.**firstmethod**(*method*)
Returns a functions that with a list of instances, finds the first instance that returns a value for the given method.

The list can also contain promises (`promise`.)

celery.utils.**fun_takes_kwargs**(*fun*, *kwlist=[]*)
With a function, and a list of keyword arguments, returns arguments in the list which the function takes.

If the object has an *argspec* attribute that is used instead of using the `inspect.getargspec()` introspection.

Parameters

- **fun** – The function to inspect arguments of.
- **kwlist** – The list of keyword arguments.

Examples

```
>>> def foo(self, x, y, logfile=None, loglevel=None):
...     return x * y
>>> fun_takes_kwargs(foo, ["logfile", "loglevel", "task_id"])
["logfile", "loglevel"]


>>> def foo(self, x, y, **kwargs):
>>> fun_takes_kwargs(foo, ["logfile", "loglevel", "task_id"])
["logfile", "loglevel", "task_id"]
```

celery.utils.**get_cls_by_name**(*name*, *aliases={}*, *imp=None*, *package=None*, ***kwargs*)
Get class by name.

The name should be the full dot-separated path to the class:

```
modulename.ClassName
```

Example:

```
celery.concurrency.processes.TaskPool
                            ^- class name
```

If *aliases* is provided, a dict containing short name/long name mappings, the name is looked up in the aliases first.

Examples:

```
>>> get_cls_by_name("celery.concurrency.processes.TaskPool")
<class 'celery.concurrency.processes.TaskPool'>
```

```
>>> get_cls_by_name("default", {
...     "default": "celery.concurrency.processes.TaskPool"})
<class 'celery.concurrency.processes.TaskPool'>
```

Does not try to look up non-string names. >>> from celery.concurrency.processes import TaskPool >>> get_cls_by_name(TaskPool) is TaskPool True

celery.utils.**get_full_cls_name**(*cls*)

With a class, get its full module and class name.

celery.utils.**get_symbol_by_name**(*name*, *aliases={}*, *imp=None*, *package=None*, ***kwargs*)

Get class by name.

The name should be the full dot-separated path to the class:

```
modulename.ClassName
```

Example:

```
celery.concurrency.processes.TaskPool
                            ^- class name
```

If *aliases* is provided, a dict containing short name/long name mappings, the name is looked up in the aliases first.

Examples:

```
>>> get_cls_by_name("celery.concurrency.processes.TaskPool")
<class 'celery.concurrency.processes.TaskPool'>


>>> get_cls_by_name("default", {
...     "default": "celery.concurrency.processes.TaskPool"})
<class 'celery.concurrency.processes.TaskPool'>
```

Does not try to look up non-string names. >>> from celery.concurrency.processes import TaskPool >>> get_cls_by_name(TaskPool) is TaskPool True

celery.utils.**import_from_cwd**(*module*, *imp=None*, *package=None*)

Import module, but make sure it finds modules located in the current directory.

Modules located in the current directory has precedence over modules located in *sys.path*.

celery.utils.**instantiate**(*name*, **args*, ***kwargs*)

Instantiate class by name.

See `get_cls_by_name()`.

celery.utils.**is_iterable**(*obj*)

celery.utils.**isatty**(*fh*)

celery.utils.**kwdict**(*kwargs*)

Make sure keyword arguments are not in unicode.

This should be fixed in newer Python versions, see: http://bugs.python.org/issue4978.

celery.utils.**lpmerge**(*L*, *R*)

Left precedent dictionary merge. Keeps values from *l*, if the value in *r* is `None`.

celery.utils.**mattrgetter**(**attrs*)

Like `operator.itemgetter()` but returns `None` on missing attributes instead of raising `AttributeError`.

celery.utils.**maybe_promise**(*value*)
Evaluates if the value is a promise.

class celery.utils.**mpromise**(*fun*, **args*, ***kwargs*)
Memoized promise.

The function is only evaluated once, every subsequent access will return the same value.

evaluated
Set to to `True` after the promise has been evaluated.

evaluate()

evaluated = False

celery.utils.**noop**(**args*, ***kwargs*)
No operation.

Takes any arguments/keyword arguments and does nothing.

celery.utils.**padlist**(*container*, *size*, *default=None*)
Pad list with default elements.

Examples:

```
>>> first, last, city = padlist(["George", "Costanza", "NYC"], 3)
("George", "Costanza", "NYC")
>>> first, last, city = padlist(["George", "Costanza"], 3)
("George", "Costanza", None)
>>> first, last, city, planet = padlist(["George", "Costanza",
                                         "NYC"], 4, default="Earth")
("George", "Costanza", "NYC", "Earth")
```

class celery.utils.**promise**(*fun*, **args*, ***kwargs*)
A promise.

Evaluated when called or if the `evaluate()` method is called. The function is evaluated on every access, so the value is not memoized (see `mpromise`).
Overloaded operations that will evaluate the promise: `__str__()`, `__repr__()`, `__cmp__()`.
evaluate()

celery.utils.**reprcall**(*name*, *args=()*, *kwargs=()*, *sep=', '*)

celery.utils.**reprkwargs**(*kwargs*, *sep=', '*, *fmt='%s=%s'*)

celery.utils.**textindent**(*t*, *indent=0*)
Indent text.

celery.utils.**truncate_text**(*text*, *maxlen=128*, *suffix='...'*)
Truncates text to a maximum number of characters.

celery.utils.**warn_deprecated**(*description=None*, *deprecation=None*, *removal=None*, *alternative=None*)

11.5.37 celery.utils.functional

- celery.utils.functional

celery.utils.functional

Utilities for functions.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

`celery.utils.functional.`**`maybe_list`**(*l*)

`celery.utils.functional.`**`memoize`**(*maxsize=None*, *Cache=<class celery.datastructures.LRUCache at 0x2efd390>*)

11.5.38 celery.utils.term

- celery.utils.term

celery.utils.term

Terminals and colors.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.utils.term.`**`colored`**(**s*, ***kwargs*)

Terminal colored text.
Example::

```
>>> c = colored(enabled=True)
>>> print(str(c.red("the quick "), c.blue("brown ", c.bold("fox ")),
...       c.magenta(c.underline("jumps over")),
...       c.yellow(" the lazy "),
...       c.green("dog ")))
```

`black`(**s*)

`blink`(**s*)

`blue`(**s*)

`bold`(**s*)

`bright`(**s*)

`cyan`(**s*)

`embed`()

`green`(**s*)

`iblue`(**s*)

`icyan`(**s*)

`igreen`(**s*)

imagenta(**s*)

ired(**s*)

iwhite(**s*)

iyellow(**s*)

magenta(**s*)

no_color()

node(*s*, *op*)

red(**s*)

reset(**s*)

reverse(**s*)

underline(**s*)

white(**s*)

yellow(**s*)

celery.utils.term.**fg**(*s*)

11.5.39 celery.utils.timeutils

- celery.utils.timeutils

celery.utils.timeutils

This module contains various utilities relating to dates and times.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

celery.utils.timeutils.**delta_resolution**(*dt*, *delta*)
Round a datetime to the resolution of a timedelta.

If the timedelta is in days, the datetime will be rounded to the nearest days, if the timedelta is in hours the datetime will be rounded to the nearest hour, and so on until seconds which will just return the original datetime.

celery.utils.timeutils.**humanize_seconds**(*secs*, *prefix=''*)
Show seconds in human form, e.g. 60 is "1 minute", 7200 is "2 hours".

celery.utils.timeutils.**maybe_iso8601**(*dt*)
Either datetime | str -> datetime or None -> None

celery.utils.timeutils.**maybe_timedelta**(*delta*)
Coerces integer to timedelta if *delta* is an integer.

celery.utils.timeutils.**rate**(*rate*)
Parses rate strings, such as *"100/m"* or *"2/h"* and converts them to seconds.

celery.utils.timeutils.**remaining**(*start*, *ends_in*, *now=None*, *relative=False*)
Calculate the remaining time for a start date and a timedelta.

e.g. "how many seconds left for 30 seconds after start?"

Parameters

- **start** – Start `datetime`.
- **ends_in** – The end delta as a `timedelta`.
- **relative** – If enabled the end time will be calculated using `delta_resolution()` (i.e. rounded to the resolution of *ends_in*).
- **now** – Function returning the current time and date, defaults to `datetime.now()`.

celery.utils.timeutils.**timedelta_seconds**(*delta*)
Convert `datetime.timedelta` to seconds.

Doesn't account for negative values.

celery.utils.timeutils.**weekday**(*name*)
Return the position of a weekday (0 - 7, where 0 is Sunday).

Example:

```
>>> weekday("sunday"), weekday("sun"), weekday("mon")
(0, 0, 1)
```

11.5.40 celery.utils.encoding

- celery.utils.encoding

celery.utils.encoding

Utilities to encode text, and to safely emit text from running applications without crashing with the infamous `UnicodeDecodeError` exception.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

celery.utils.encoding.**bytes_to_str**(*s*)

celery.utils.encoding.**default_encoding**()

celery.utils.encoding.**ensure_bytes**(*s*)

celery.utils.encoding.**from_utf8**(*s*, **args*, ***kwargs*)

celery.utils.encoding.**safe_repr**(*o*, *errors='replace'*)

celery.utils.encoding.**safe_str**(*s*, *errors='replace'*)

celery.utils.encoding.**str_to_bytes**(*s*)

11.5.41 celery.utils.compat

- celery.utils.compat

celery.utils.compat

Backward compatible implementations of features only available in newer Python versions.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

`celery.utils.compat.`**`chain_from_iterable`**`()`
chain.from_iterable(iterable) –> chain object

Alternate chain() contructor taking a single iterable argument that evaluates lazily.

11.5.42 celery.utils.patch

- celery.utils.patch

celery.utils.patch

Monkey-patch to ensure loggers are process aware.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

`celery.utils.patch.`**`ensure_process_aware_logger`**`()`

11.5.43 celery.utils.serialization

- celery.utils.serialization

celery.utils.serialization

Utilities for safely pickling exceptions.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

exception celery.utils.serialization.**UnpickleableExceptionWrapper**(*exc_module*, *exc_cls_name*, *exc_args*, *text=None*)

Wraps unpickleable exceptions.

Parameters

- **exc_module** – see exc_module.
- **exc_cls_name** – see exc_cls_name.
- **exc_args** – see exc_args

Example

```
>>> try:
...     something_raising_unpickleable_exc()
>>> except Exception, e:
...     exc = UnpickleableException(e.__class__.__module__,
...                                 e.__class__.__name__,
...                                 e.args)
...     pickle.dumps(exc) # Works fine.
```

exc_args = None

The arguments for the original exception.

exc_cls_name = None

The name of the original exception class.

exc_module = None

The module of the original exception.

classmethod from_exception(*exc*)

restore()

celery.utils.serialization.**create_exception_cls**(*name*, *module*, *parent=None*)

Dynamically create an exception class.

celery.utils.serialization.**find_nearest_pickleable_exception**(*exc*)

With an exception instance, iterate over its super classes (by mro) and find the first super exception that is pickleable. It does not go below Exception (i.e. it skips Exception, BaseException and object). If that happens you should use UnpickleableException instead.

Parameters **exc** – An exception instance.

Returns the nearest exception if it's not Exception or below, if it is it returns None.

:rtype Exception:

celery.utils.serialization.**get_pickleable_exception**(*exc*)

Make sure exception is pickleable.

celery.utils.serialization.**get_pickled_exception**(*exc*)

Get original exception from exception pickled using get_pickleable_exception().

celery.utils.serialization.**subclass_exception**(*name*, *parent*, *module*)

celery.utils.serialization.**unwanted_base_classes = (<type 'exceptions.StandardError'>, <type 'exceptions.E**

List of base classes we probably don't want to reduce to.

11.5.44 celery.utils.timer2

- timer2

timer2

Scheduler for Python functions.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class celery.utils.timer2.**Entry**(*fun*, *args=None*, *kwargs=None*)

cancel()

cancelled = False

class celery.utils.timer2.**Schedule**(*max_interval=2*, *on_error=None*)
ETA scheduler.

clear()

empty()
Is the schedule empty?

enter(*entry*, *eta=None*, *priority=0*)
Enter function into the scheduler.

Parameters

- **entry** – Item to enter.
- **eta** – Scheduled time as a `datetime.datetime` object.
- **priority** – Unused.

handle_error(*exc_info*)

info()

on_error = None

queue

exception celery.utils.timer2.**TimedFunctionFailed**

class celery.utils.timer2.**Timer**(*schedule=None*, *on_error=None*, *on_tick=None*, ***kwargs*)

class Entry(*fun*, *args=None*, *kwargs=None*)

cancel()

cancelled = False

class Timer.**Schedule**(*max_interval=2*, *on_error=None*)
ETA scheduler.

clear()

empty()
Is the schedule empty?

enter(*entry*, *eta=None*, *priority=0*)
Enter function into the scheduler.

Parameters

- **entry** – Item to enter.

- **eta** – Scheduled time as a `datetime.datetime` object.
- **priority** – Unused.

handle_error(*exc_info*)

info()

on_error = None

queue

Timer.**apply_after**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Timer.**apply_at**(*eta*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Timer.**apply_entry**(*entry*)

Timer.**apply_interval**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Timer.**cancel**(*tref*)

Timer.**clear**()

Timer.**empty**()

Timer.**ensure_started**()

Timer.**enter**(*entry*, *eta*, *priority=None*)

Timer.**enter_after**(*msecs*, *entry*, *priority=0*)

Timer.**exit_after**(*msecs*, *priority=10*)

Timer.**next**()

Timer.**on_tick** = None

Timer.**queue**

Timer.**run**()

Timer.**running** = False

Timer.**stop**()

celery.utils.timer2.**to_timestamp**(*d*)

11.5.45 celery.utils.dispatch

11.5.46 celery.utils.dispatch.signal

Signal class.

class celery.utils.dispatch.signal.**Signal**(*providing_args=None*)

Base class for all signals

receivers
Internal attribute, holds a dictionary of
`{receiverkey (id): weakref(receiver)}` mappings.

connect(*receiver*, *sender=None*, *weak=True*, *dispatch_uid=None*)

Connect receiver to sender for signal.

Parameters

- **receiver** – A function or an instance method which is to receive signals. Receivers must be hashable objects.

  if weak is `True`, then receiver must be weak-referencable (more precisely `saferef.safe_ref()` must be able to create a reference to the receiver).

  Receivers must be able to accept keyword arguments.

  If receivers have a *dispatch_uid* attribute, the receiver will not be added if another receiver already exists with that *dispatch_uid*.
- **sender** – The sender to which the receiver should respond. Must either be of type `Signal`, or `None` to receive events from any sender.
- **weak** – Whether to use weak references to the receiver. By default, the module will attempt to use weak references to the receiver objects. If this parameter is false, then strong references will be used.
- **dispatch_uid** – An identifier used to uniquely identify a particular instance of a receiver. This will usually be a string, though it may be anything hashable.

disconnect(*receiver=None*, *sender=None*, *weak=True*, *dispatch_uid=None*)

Disconnect receiver from sender for signal.

If weak references are used, disconnect need not be called. The receiver will be removed from dispatch automatically.

Parameters

- **receiver** – The registered receiver to disconnect. May be none if *dispatch_uid* is specified.
- **sender** – The registered sender to disconnect.
- **weak** – The weakref state to disconnect.
- **dispatch_uid** – the unique identifier of the receiver to disconnect

send(*sender*, ***named*)

Send signal from sender to all connected receivers.

If any receiver raises an error, the error propagates back through send, terminating the dispatch loop, so it is quite possible to not have all receivers called if a raises an error.

Parameters

- **sender** – The sender of the signal. Either a specific object or `None`.
- ****named** – Named arguments which will be passed to receivers.

Returns a list of tuple pairs: *[(receiver, response), ...]*.

send_robust(*sender*, ***named*)

Send signal from sender to all connected receivers catching errors.

Parameters

- **sender** – The sender of the signal. Can be any python object (normally one registered with a connect if you actually want something to occur).
- ****named** – Named arguments which will be passed to receivers. These arguments must be a subset of the argument names defined in `providing_args`.

Returns a list of tuple pairs: *[(receiver, response), ...]*.

Raises DispatcherKeyError

if any receiver raises an error (specifically any subclass of `Exception`), the error instance is returned as the result for that receiver.

11.5.47 celery.utils.dispatch.saferef

"Safe weakrefs", originally from pyDispatcher.

Provides a way to safely weakref any function, including bound methods (which aren't handled by the core weakref module).

class celery.utils.dispatch.saferef.**BoundMethodWeakref**(*target*, *on_delete=None*)
'Safe' and reusable weak references to instance methods.

BoundMethodWeakref objects provide a mechanism for referencing a bound method without requiring that the method object itself (which is normally a transient object) is kept alive. Instead, the BoundMethodWeakref object keeps weak references to both the object and the function which together define the instance method.

key
the identity key for the reference, calculated by the class's `calculate_key()` method applied to the target instance method

deletion_methods
sequence of callable objects taking single argument, a reference to this object which will be called when *either* the target object or target function is garbage collected (i.e. when this object becomes invalid). These are specified as the on_delete parameters of `safe_ref()` calls.

weak_self
weak reference to the target object

weak_func
weak reference to the target function

_all_instances
class attribute pointing to all live BoundMethodWeakref objects indexed by the class's *calculate_key(target)* method applied to the target objects. This weak value dictionary is used to short-circuit creation so that multiple references to the same (object, function) pair produce the same BoundMethodWeakref instance.

classmethod calculate_key(*target*)
Calculate the reference key for this reference

Currently this is a two-tuple of the *id()*'s of the target object and the target function respectively.

class celery.utils.dispatch.saferef.**BoundNonDescriptorMethodWeakref**(*target*, *on_delete=None*)
A specialized `BoundMethodWeakref`, for platforms where instance methods are not descriptors.

It assumes that the function name and the target attribute name are the same, instead of assuming that the function is a descriptor. This approach is equally fast, but not 100% reliable because functions can be stored on an attribute named differenty than the function's name such as in:

```
>>> class A(object):
...     pass

>>> def foo(self):
...     return "foo"
>>> A.bar = foo
```

But this shouldn't be a common use case. So, on platforms where methods aren't descriptors (such as Jython) this implementation has the advantage of working in the most cases.

celery.utils.dispatch.saferef.**get_bound_method_weakref**(*target*, *on_delete*)
Instantiates the appropiate `BoundMethodWeakRef`, depending on the details of the underlying class method implementation.

celery.utils.dispatch.saferef.**safe_ref**(*target*, *on_delete=None*)
Return a *safe* weak reference to a callable target

Parameters

- **target** – the object to be weakly referenced, if it's a bound method reference, will create a `BoundMethodWeakref`, otherwise creates a simple `weakref.ref`.

- **on_delete** – if provided, will have a hard reference stored to the callable to be called after the safe reference goes out of scope with the reference object, (either a `weakref.ref` or a `BoundMethodWeakref`) as argument.

11.5.48 celery.platforms

- celery.platforms

celery.platforms

Utilities dealing with platform specifics: signals, daemonization, users, groups, and so on.

copyright

3. 2009 - 2011 by Ask Solem.

license BSD, see LICENSE for more details.

class `celery.platforms.`**`DaemonContext`**(*pidfile=None*, *workdir=None*, *umask=None*, ***kwargs*)

`close`(**args*)

`open`()

`umask` = **0**

`workdir` = **'/'**

exception `celery.platforms.`**`LockFailed`**
Raised if a pidlock can't be acquired.

class `celery.platforms.`**`PIDFile`**(*path*)
PID lock file.

This is the type returned by `create_pidlock()`.

Should not be used directly, use the :func:'create_pidlock' context instead

`acquire`()
Acquire lock.

`is_locked`()
Returns true if the pid lock exists.

`path` = **None**
Path to the pid lock file.

`read_pid`()
Reads and returns the current pid.

`release`(**args*)
Release lock.

`remove`()
Removes the lock.

`remove_if_stale`()
Removes the lock if the process is not running. (does not respond to signals).

write_pid()

class celery.platforms.**Signals**
Convenience interface to signals.

If the requested signal is not supported on the current platform, the operation will be ignored.

Examples:

```
>>> from celery.platforms import signals

>>> signals["INT"] = my_handler

>>> signals["INT"]
my_handler

>>> signals.supported("INT")
True

>>> signals.signum("INT")
2

>>> signals.ignore("USR1")
>>> signals["USR1"] == signals.ignored
True

>>> signals.reset("USR1")
>>> signals["USR1"] == signals.default
True

>>> signals.update(INT=exit_handler,
...                TERM=exit_handler,
...                HUP=hup_handler)
```

default = 0

ignore(**signal_names*)
Ignore signal using SIG_IGN.

Does nothing if the platform doesn't support signals, or the specified signal in particular.

ignored = 1

reset(**signal_names*)
Reset signals to the default signal handler.

Does nothing if the platform doesn't support signals, or the specified signal in particular.

signum(*signal_name*)
Get signal number from signal name.

supported(*signal_name*)
Returns true value if signal_name exists on this platform.

update(*_d_=None*, ***sigmap*)
Set signal handlers from a mapping.

celery.platforms.**create_pidlock**(*pidfile*)
Create and verify pid file.

If the pid file already exists the program exits with an error message, however if the process it refers to is not running anymore, the pid file is deleted and the program continues.

The caller is responsible for releasing the lock before the program exits.

Returns PIDFile.

Example:

```
import atexit
pidlock = create_pidlock("/var/run/app.pid").acquire()
atexit.register(pidlock.release)
```

celery.platforms.**detached**(*logfile=None*, *pidfile=None*, *uid=None*, *gid=None*, *umask=0*, *workdir=None*, ***opts*)

Detach the current process in the background (daemonize).

Parameters

- **logfile** – Optional log file. The ability to write to this file will be verified before the process is detached.
- **pidfile** – Optional pid file. The pid file will not be created, as this is the responsibility of the child. But the process will exit if the pid lock exists and the pid written is still running.
- **uid** – Optional user id or user name to change effective privileges to.
- **gid** – Optional group id or group name to change effective privileges to.
- **umask** – Optional umask that will be effective in the child process.
- **workdir** – Optional new working directory.
- ****opts** – Ignored.

Example:

```
import atexit
from celery.platforms import detached, create_pidlock

with detached(logfile="/var/log/app.log", pidfile="/var/run/app.pid",
              uid="nobody"):
    # Now in detached child process with effective user set to nobody,
    # and we know that our logfile can be written to, and that
    # the pidfile is not locked.
    pidlock = create_pidlock("/var/run/app.pid").acquire()
    atexit.register(pidlock.release)

    # Run the program
    program.run(logfile="/var/log/app.log")
```

celery.platforms.**get_fdmax**(*default=None*)

Returns the maximum number of open file descriptors on this system.

Parameters **default** – Value returned if there's no file descriptor limit.

celery.platforms.**parse_gid**(*gid*)

Parse group id.

gid can be an integer (gid) or a string (group name), if a group name the gid is taken from the password file.

celery.platforms.**parse_uid**(*uid*)

Parse user id.

uid can be an integer (uid) or a string (user name), if a user name the uid is taken from the password file.

celery.platforms.**pyimplementation**()

celery.platforms.**set_effective_user**(*uid=None*, *gid=None*)

Change process privileges to new user/group.

If UID and GID is set the effective user/group is set.

If only UID is set, the effective user is set, and the group is set to the users primary group.

If only GID is set, the effective group is set.

celery.platforms.**set_mp_process_title**(*progname*, *info=None*, *hostname=None*)
Set the ps name using the multiprocessing process name.

Only works if `setproctitle` is installed.

celery.platforms.**set_process_title**(*progname*, *info=None*)
Set the ps name for the currently running process.

Only works if `setproctitle` is installed.

celery.platforms.**setegid**(*gid*)
Set effective group id.

celery.platforms.**seteuid**(*uid*)
Set effective user id.

celery.platforms.**setgid**(*gid*)

celery.platforms.**setuid**(*uid*)

celery.platforms.**shellsplit**(*s*, *posix=True*)

celery.platforms.**strargv**(*argv*)

CHAPTER 12

Indices and tables

- *genindex*
- *modindex*
- *search*

C

celery.app, ??
celery.app.amqp, ??
celery.app.defaults, ??
celery.app.task, ??
celery.apps.beat, ??
celery.apps.worker, ??
celery.backends, ??
celery.backends.amqp, ??
celery.backends.base, ??
celery.backends.cache, ??
celery.backends.cassandra, ??
celery.backends.database, ??
celery.backends.mongodb, ??
celery.backends.redis, ??
celery.backends.tyrant, ??
celery.beat, ??
celery.bin.base, ??
celery.bin.camqadm, ??
celery.bin.celerybeat, ??
celery.bin.celeryctl, ??
celery.bin.celeryd, ??
celery.bin.celeryd_multi, ??
celery.bin.celeryev, ??
celery.concurrency, ??
celery.concurrency.base, ??
celery.concurrency.eventlet, ??
celery.concurrency.gevent, ??
celery.concurrency.processes, ??
celery.concurrency.processes.pool, ??
celery.concurrency.solo, ??
celery.concurrency.threads, ??
celery.contrib.abortable, ??
celery.contrib.batches, ??
celery.contrib.rdb, ??
celery.datastructures, ??
celery.db.models, ??
celery.db.session, ??
celery.events, ??
celery.events.cursesmon, ??
celery.events.dumper, ??
celery.events.snapshot, ??
celery.events.state, ??
celery.exceptions, ??
celery.execute.trace, ??
celery.loaders, ??
celery.loaders.app, ??
celery.loaders.base, ??
celery.loaders.default, ??
celery.log, ??
celery.platforms, ??
celery.registry, ??
celery.result, ??
celery.routes, ??
celery.schedules, ??
celery.signals, ??
celery.states, ??
celery.task, ??
celery.task.base, ??
celery.task.chords, ??
celery.task.control, ??
celery.task.http, ??
celery.task.sets, ??
celery.utils, ??
celery.utils.compat, ??
celery.utils.dispatch, ??
celery.utils.dispatch.saferef, ??
celery.utils.dispatch.signal, ??
celery.utils.encoding, ??
celery.utils.functional, ??
celery.utils.mail, ??
celery.utils.patch, ??
celery.utils.serialization, ??
celery.utils.term, ??
celery.utils.timer2, ??
celery.utils.timeutils, ??
celery.worker, ??
celery.worker.autoscale, ??
celery.worker.buckets, ??
celery.worker.consumer, ??
celery.worker.heartbeat, ??

celery.worker.job, ??
celery.worker.mediator, ??
celery.worker.state, ??